buildout.coredev Documentation

Release 4.3

The Plone Foundation

September 24, 2015

Contents

1 Introduction **1**
1.1 STOP! . . . 1
1.2 Table of Awesome . . . 1
1.3 Others . . . 18
1.4 Translations available . . . 22

CHAPTER 1

Introduction

This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

1.1 STOP!

Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

1.2 Table of Awesome

1.2.1 Contributing to Plone Core

There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If

you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub

Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

1.2.2 Plone Developer Culture

If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

1.2.3 Getting started with development

This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy

If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment

The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches

If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing

Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally

In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg

Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests

Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone

If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone

If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My!

Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review

- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins

You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets

If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

1.2.4 Writing documentation

Documentation of Plone

As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual

  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics

At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

1.2.5 Style Guide

Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions

Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions

In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

1.2.6 Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP

Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs

After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP

You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch

Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage
```

```
[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up

Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

1.2.7 Troubleshooting

Buildout Issues

Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py

You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy

`mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors

```
ERROR: You are not in a path which has mr.developer installed
(:file:`.mr.developer.cfg` not found).
```

When running any **./bin/develop** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages

"ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **bin/develop update --force**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2

```
ImportError:  No module named Zope2" when building using a PLIP cfg file.
```

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py'

Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **bin/buildout ...** after **bin/develop ...**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL

`pil.cfg` is include within this buildout to aid in PIL installation. Run **bin/buildout -c pil.cfg** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues

bin/develop status is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

1.2.8 How to Update these Docs

These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

1.3 Others

These are some documents using as reference for this documentation.

1.3.1 Contributor's Agreement for Plone Explained

Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement

The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers

What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

1.3.2 Mr. Developer

This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

1.3.3 Reviewing PLIPs

Expectations

A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment

Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review

There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

1.4 Translations available

There are some initiatives to maintenance available translations in other languages for this documentation like:

1.4.1 Procesos para el desarrollo del núcleo de Plone

Introducción

Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE!

Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.

- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código

- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.

  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de

un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a

empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros

Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.

3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL

podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles

Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:'.mr.developer.cfg' not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:'.mr.developer.cfg' not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 550
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:'.mr.developer.cfg' not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.

- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos

paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los

revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.
- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio

github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

CHANGES.txt El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8. La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el

PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.

7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?

- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

> Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

> A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

> Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

> No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

> Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

> Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

> El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

> Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?

- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical

purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.
3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Procesos para el desarrollo del núcleo de Plone

Introducción Esta documentación describes el proceso para desarrollar en Plone. Este es principalmente un recurso técnico para instalar su configuración buildout del desarrollador del núcleo del CMS, corregir fallos, y escribir plips.

¡DETÉNGASE! Legalmente, usted NO puede contribuir al código al menos que usted allá firmado el acuerdo de contribución. Esto significa que nosotros NO podemos aceptar las peticiones de mejoras de usted pueda presentar a no ser que allá hecho cumplido con este requisito, por favor, no ponga código fuente para ser revisado de ninguna forma si no ha hecho eso. Presentar el acuerdo de contribución es fácil (y pronto será más fácil) y si quieres acceso rápido y esta familiarizado con comunidad, entre dentro del canal irc y preguntar por los administradores del repositorio para otorgarle acceso en base a una versión digitalizada del acuerdo. ¡Ellos lo harán lo mas rápido posible!

Tabla de contenidos

Contribuyendo al núcleo de Plone Hay muchas personas y compañías quiénes confían en Plone diariamente, por tanto nosotros tenemos que introducir algún nivel de control en la calidad del código fuente. Todo el código de fuente de Plone es hospedado en un repositorio git en https://github.com/plone, solo para los miembros del equipo de desarrollado que tienen derecho de hacer revisiones.

El enviar el acuerdo de contribuciones (contributors agreement) no garantiza que se otorguen acceso al repositorio, pero una vez que lo envía siempre lo tenemos en el archivo para cuando usted está listo para contribuir. Nosotros pedimos que antes de solicitar el acceso de escritura al núcleo de Plone, usted debe familiarizarse un poco con la comunidad, ya que le ayudará a acelerar su iniciación:

- Preguntar y (especialmente) responder a preguntas en el sitio web stack overflow y canal IRC con un enfoque para conocer un poco a los desarrolladores activos.
- Asistir una conferencia / un simposio o participar en un sprints / tune-ups. Hay abundancia de oportunidades de conocer la comunidad y comenzar a contribuir a través de varias sesiones de programación, ya sea en persona o en la web. Usted puede incluso ser capaz de obtener acceso inmediato en una conferencia si usted ha demostrando sus habilidades de programación y las personas adecuada asisten.
- Puede iniciarse en la contribuciones en el repositorio de collective. No se preocupe por hacer todo perfecto o no complique al pedir ayuda, de esta forma usted nos hace saber a nosotros como mejorar nuestro código juntos como una comunidad.
- **Parches / Patches:** Históricamente nosotros animamos a las personas para entregar parches *(patches)* al colector del ticket. Estos tickets suelen ser ignorados para siempre. Técnicamente, para que nosotros aceptemos su parche usted debe firmar el *contributor agreement*. Si usted quiere contribuir a corregir del código, por favor sólo debe firmar el acuerdo y pasar por el proceso estándar github de pull request descrito hasta que se sienta cómodo para pasar revisión. Si el ticket es trivial, o está arreglando la documentación, no es necesario firmar el *contributor agreement*.

Una vez te se allá familiarizado usted mismo con la comunidad y usted está entusiasmado para contribuir al núcleo:

- Firme y envié un correo con el contributor agreement ubicado en http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, entonces sea bien por correo postal a la dirección indicada o escanear y enviar por correo electrónico a assignments@plone.org. Esto ofrece protección de derechos de autor y se asegura de que la Fundación Plone que sea capaz de ejercer un cierto control sobre el código base, asegurándose de que no es apropiado para fines poco éticos de alguien. Para preguntas acerca de por qué el acuerdo está requerido, por favor ver Contributor's Agreement for Plone Explained.

Si usted no esta seguro por dónde empezar o solo querer más dirección, siéntase libre de usar el canal IRC, las listas de correo, Twitter, etc... y pedir ayuda. Si bien no existe un proceso oficial de tutoría, hay un montón de personas dispuestas a actuar en ese rol y le guiará por los pasos de involucrarse en la comunidad. Una manera común de empezar las contribuciones es participar en un día de tune-up de Plone. Los Tune-ups está llenado con una buena mezcla de programadores recién iniciados y experimentados igualmente. Para más información, por favor ver http://plone.org/tuneup.

¡Sea bienvenido a la comunidad Plone!

Lidiar con los pull requests en GitHub Antes de que podamos `merge` un `pull request`, tenemos que comprobar que el autor ha firmado el acuerdo del contribuyente.

Si están listados en https://github.com/plone?tab=members, el autor ha firmado para que podamos seguir adelante y hacer un `merge`.

Si no están en la lista, todavía hay una posibilidad de que han firmado el acuerdo del contribuyente. Verifique eso en el canal IRC #plone-framework.

Los `pull requests` sin el acuerdo del contribuyente sólo puede ser fusionadas en casos triviales, y sólo por el encargado de la liberación de Plone o *release manager*.

Cultura de desarrollo en Plone Si usted va a contribuir algo a Plone, pedimos un par de cosas. En primer lugar, por favor, únase a la lista de correo de desarrolladores de Plone plone-developers y, como mínimo, para estar al acecho de los acontecimientos alrededor de esta lista. Usted pronto vera cómo la gente trabaja y qué tipo de cosas son las más adecuadas para la discusión grupal. Segundo, por favor, pida ayuda para configurar su entorno en el canal IRC. La mayoría de nuestros desarrolladores trabajan allí y obtendrá el mejor consejo desde allí.

Descargue un software de cliente IRC (o use un cliente alternativo a través de la web) y entre dentro de la sala IRC #plone-framework (y / o en la sala #plone - ambos en el servidor freenode). La gente en la sala IRC #plone-framework tienen mucho tiempo usando Plone y por eso estarán muy felices en ayudarle a ir por la dirección correcta y hacerle tomar las decisiones mas adecuadas. Mas información sobre el servicio de IRC puede ser encontrada en el siguiente enlace http://plone.org/support/chat.

Si usted está activamente suministrando código, únase la lista de correo electrónico del bot de prueba así usted sabe si sus recientes commits ha dañado algo (o corregirlo!) la construcción.

Tenga en cuenta que es posible que usted este en una zona horaria distinta cuándo necesita hacer preguntas en el canal IRC y las cosas no están muy activas, por favor, escriba un mensaje a la lista de correo plone-developers o tome una cerveza y relájese, espere a que la gente se despierte del otro lado del computador o del mundo.

Cuando tenga dudas, por favor, pregunte. El código base es muy complicado y todo el mundo se confiere de que lo que sucede es lo correcto. A pesar del mal humor ocasional aquí y allá, la mayoría de los desarrolladores de Plone se salen de su camino para ayudarle a que usted consiga la ruta correcta.

Como presentar corrección de fallos al núcleo de Plone Este documento supone que usted quiere corregir un fallo y detallará el proceso completo para hacerlo. Para más información en la redacción de PLIPS, por favor valla aquí.

Política sobre soporte a la Versión Si usted está tratando de corregir fallos, tenga en cuenta que Plone tiene una política sobre soporte al versionamiento.

Dependencias

- Git
- Subversion
- Python 2.6 o 2.7 incluyendo sus librerías de encabezados, usadas para el desarrollo.
- Si usted esta usando un Mac OSX, necesitara instalar XCode. Usted puede hacer este a través de la app store o muchos otros métodos en que lo venden. Usted probablemente querrá instalar su propio python 2.6 entonces necesita fuera todos los archivos encabezados de las librerías cuál hace compilar algunas extensiones. Usted puede ignorar este consejo para empezar, pero créame, que usted tendrá que instalarlas mas adelante. Ellos siempre hacen...
- Python Imaging Library (PIL). Debe asegurarse instalar esto dentro del entorno apropiado de python.
- VirtualEnv en el apropiado entorno python.
- GCC para compilar ZODB, Zope y lxml.
- libxml2 y libxslt, incluyendo sus librerías de encabezados para desarrollo.

Instalando su Entorno de Desarrollo El primer paso en corregir un fallo está consiguiendo ejecutar este buildout correctamente. Le recomendamos corregir un fallo en la branch más reciente y entonces backporting tan como sea necesario. Github por defecto siempre apunta a la branch actualmente activa. Más información sobre el cambio de las branches de liberación a continuación.

Para instalar un entorno de desarrollo plone 4.2 debe ejecutar los siguientes comandos:

```
> cd ~/buildouts # o donde tu quieras colocar sus cosas
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (donde "python" es el ejecutable de su interprete python
> bin/buildout -v
```

Si se encuentra con problemas en este proceso, consulte la documentación Troubleshooting: solucionar problemas.

Esto se ejecutará durante mucho tiempo, si es su primera ejecución de buildout (~ 20 minutos) o tal ves mas todo dependerá de su conexión a Internet. Una vez que se hace ejecutado el buildout, usted puede iniciar su nueva instancia Zope con el siguiente comando:

```
> ./bin/instance fg
```

El usuario y contraseña por defecto para una instancia Zope de desarrollo es **admin/admin**.

Cambiar Branches Si su fallo es especifico en una branch o usted piensa que debería hacer backport, usted puede cambiar fácilmente las branches. La primera ves usted tiene que obtener una branch, entonces usted debe hacer eso con el siguiente comando:

```
> git checkout -t origin/4.1
```

Esto debería crear una branch local de 4.1 con el seguimiento de la una en el servicio github. A partir de entonces sólo se puede hacer con el siguiente comando:

```
> git checkout 4.1
```

Para ver en qué branch usted se encuentra actualmente, solo ejecute el siguiente comando:

```
> git branch
```

La línea con un * por él indicará qué branch actualmente estás trabajando.

Important: ¡Asegúrese de volver a ejecutar buildout si estuviera en una branch diferente antes para obtener las versiones correctas de los paquetes, de lo contrario obtendrá un comportamiento extraño!

Para mas información sobre buildout, por favor ver la documentación sobre buildout en el manual de desarrollador collective.

Comprobando paquetes para corregir La mayoría de paquetes no están por defecto en el directorio `src/`, así que usted puede usar `mr.developer` para conseguir la versión mas reciente y asegurarse que usted siempre tiene la versión mas actualizada. Puede ser un poco intimidado al principio para averiguar qué paquetes están causando el fallo en cuestión, pero sólo pregunte en el IRC si necesitas algo de ayuda. Una vez que [usted cree] que sabe cuál(es) paquete(s) usted que desea, nosotros tenemos que obtener de la fuente del mismo.

Usted puede conseguir el código fuente del paquete con `mr.developer` o con el comando `checkout`, o puedes ir directamente a editar `checkouts.cfg`. Nosotros recomendamos el último pero describirá ambos. Al final, el archivo `checkouts.cfg` tiene que ser configurado de cualquier manera así que también podrás empezar allí.

En el directorio raíz de su su buildout, abra el archivo `checkouts.cfg` y añada su paquete si no es ya allí de la siguiente forma:

```
auto-checkout =
        # mi paquetes modificados
        plone.app.caching
        plone.caching
```

```
        # otros paquetes
        ...
```

Entonces ejecutar de nuevo buildout para conseguir los códigos fuentes de los paquetes desde sus repositorios, con el siguiente comando:

```
> ./bin/buildout
```

Alternativamente, nosotros podemos administrar los checkouts desde la línea de comando, usando el comando de mr.developer **`bin/develop`** para conseguir la fuente de paquetes más reciente. Por ejemplo, si la incidencia esta en los paquetes `plone.app.caching` y `plone.caching` lo ejecuta con los siguientes comandos:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

¡No olvide volver a ejecutar buildout! En ambos métodos, `mr.developer` descargará el código fuente desde github (o de donde se definió) y colocara el paquete en el directorio `src/`. Usted puede repetir este proceso con tantos paquetes cuando lo necesite. Para algunos más consejos en la forma de trabajo con `mr.developer`, por favor lea mas aquí.

Probando localmente En un mundo ideal, usted debería escribir un caso de prueba para su incidencia antes de tratar de corregir el fallo. En realidad esto rara ves sucede. No importa la forma cómo usted te acercas a resolver la falla, usted tiene que SIEMPRE probar la ejecución de los casos de prueba para ambos el módulo y plone.org antes de que usted genere una revisión con cualquiera de cambios.

¡Si usted no comienza con un caso de prueba, se ahorrará problemas potenciales y valida el fallo antes de llegar demasiado profundo en la incidencia!

Para correr una prueba para el módulo específico ejecute el siguiente comando:

```
> ./bin/test -m plone.app.caching
```

Estos deberían ejecutarse todo sin fallos. ¡Por favor, no verifique nada adicional! Si usted no ha escrito su caso de prueba aun, este es un buen momento para escribir un caso de prueba para la falla que usted está reparando y asegúrese de que todo está funcionando como debería.

Después que las pruebas al nivel del módulo se ejecutan con su cambio realizado, por favor asegúrese de que los otros módulos no se ven afectados por el cambio realizado por usted, para esto ejecute todas las pruebas con el siguiente comando:

```
> ./bin/alltests
```

Note: Las pruebas toman un tiempo en ejecutarse. Una ves se allá convertido en el maestro de corrección de fallas, usted tal ves solo le deje hacer esto al servicio de `jenkins` hacer esta tarea por usted. Más sobre esto a continuación.

Actualizar el archivo CHANGES.rst y checkouts.cfg Una ves todo las pruebas se ejecuten localmente en su maquina, usted debe estar **CASI** listo para generar una revisión de sus cambios. Un par de cosas hay que hacer antes de continuar.

Lo primero, por favor, edite el archivo `CHANGES.rst` (o `CHANGES.txt`, o `HISTORY.txt`) en cada archivo que usted modifico y agregue un resumen de sus cambios. En esta nota el cambio será cotejada para la próxima versión Plone y es importante para los integradores y desarrolladores puedan ser capaz de ver lo que obtendrán si se actualizan. Nuevas entradas al changelog debería ser agregadas en la partes superiores del archivo `CHANGES.txt`.

Lo más importante, si no lo hizo antes, edite el archivo `checkouts.cfg` en el directorio de buildout y agregar el paquete al cual le hizo sus cambios a la lista de `auto-checkout`. Esto le permite al release manager de Plone saber

que paquete ha sido actualizado para que cuando se de la próxima versión de Plone, este tendrá que fijar a la próxima versión del paquete al momento de generar un nuevo paquete Egg. LEER: esto es como su corrección viene en un paquete egg!

Tenga en cuenta que hay una separador de sección llamada `# Test Fixes Only`. Asegúrese que su paquete egg este por encima de esa línea o su paquete egg probablemente no se hizo muy rápidamente. Esto le dice al release manager que los paquetes Egg por debajo de esta línea tienen pruebas que están actualizadas, pero no hay cambios en el código.

Modifique el archivo `checkouts.cfg` también ejecute el comando buildbot, entonces el servicio jenkins, actualizara el paquete egg y ejecutara todas las pruebas contra las pruebas que usted realizo. No sea que usted alguna vez volvería a sáltate ejecutar todas las pruebas, por supuesto... Más sobre esto a continuación.

Si su fallo esta en mas de una publicación (ej. 4.1 y 4.2), por favor, aplicar sus cambios en ambas branches y añadir al archivo `checkouts.cfg`.

Generando una revisión y haciendo Pull Requests ¡Uf! Estamos en la recta final. Verifique su lista de actividades hechas en los últimos minutos:

- ¿Usted corrigió el fallo original?
- ¿Su código es consistente con nuestra Guía de estilos?
- ¿Usted removió lineas extras de código y PDB persistentes?
- ¿Usted escribió un caso de prueba para su fallo?
- ¿Todos sus casos de prueba para los módulos y para Plone se ejecutan sin ningún problema?
- ¿Usted actualizo el archivo `CHANGES.rst` en cada paquete que usted modifico?
- ¿Usted añadió sus paquetes cambiados al archivo `checkouts.cfg`?

Si usted respondió *SI* a todas estas preguntas, ¡usted esta listo para presentar sus cambios! Un par de recordatorios rápidos:

- Solamente generar una revisión directamente a la branch de desarrollo, si usted esta seguro que su código no causa ninguna falla y los cambios son pequeños y triviales. De lo contrario, por favor, haga un `fork` del repositorio aplicando sus revisiones allí y luego haga un `pull request` (mas abajo se explica como).
- Por favor, trate de hacer un cambio por cada revisión. Si usted esta corrigiendo tres fallas, haga tres revisiones. De esta forma, es fácil ver que fue cambiado y donde se realizo el cambio, además es mas fácil hacer un `roll back` de cualquier cambio si es necesario. Si usted quiere hacer muchos cambios como limpiar espacios en blanco o renombrar variables, es especialmente importante hacer una revisión separada por esta razón.
- Nosotros tenemos un grupo de guardianes que siguen los cambios y cada revisión aplicada para ver que ha sucedido de nuevo en el código fuente de nuestro favorito CMS! Si su revisión tiene algo REALMENTE no está bien, ellos le contactaran políticamente a usted, lo mas común que suceda es que inmediatamente revierten los cambios aplicados con sus revisiones. Hay personas no oficiales asignadas a esto si usted esta especialmente nervioso, entre en el canal IRC #plone en freenode.net y pregunte por alguien que pueda ver sus cambios.

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión del commit (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Generando revisiones al paquete Products.CMFPlone Si usted esta trabajando un corregir un fallo en el paquete `Products.CMFPlone`, hay un par de otras cosas que debe tomar en cuenta. Primero y mas importante, puede ver que este paquete tiene varias branches. Al momento de escribir este documento, habían tres branches para `4.1`, `4.2`, y la `master`, el cual es implícitamente 4.3.

¿Aun me sigue con la explicación? Entonces, usted tiene un corrección de fallas para 4.x. Si la corrección es solamente para una versión, asegúrese de obtener la branch y aplicar sus cambios allí. Sin embargo, si la corrección del fallo es en múltiples branches.

Por ejemplo, el fallo inicia en la versión 4.1. Obtenga la branch 4.1 y aplicar sus cambios allí con varias revisiones por cada cambio con sus respectivas pruebas.

Si su corrección involucra una simple revisión de cambios, usted puede usar el comando `git cherry-pick` para aplicar la misma revisión a un branch diferente.

Primero cambie a la branch, con el siguiente comando:

```
> git checkout 4.2
```

Y entonces con el comando `git cherry-pick` y el número de revisión (usted puede obtener el número SHA hash desde el `git log`), con el siguiente comando:

```
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando `git cherry-pick`.

Si su corrección involucra múltiples revisiones, entonces hacer un `cherry-picking` uno a uno puede resultar tedioso. En este caso, las cosas son más fáciles, si usted hizo su corrección en una branch con una característica separada.

En ese escenario, primero fusione la branch característica a la branch 4.1, con los siguientes comandos:

```
> git checkout 4.1
> git merge my-awesome-feature
```

A continuación, regrese a la branch característica y haga una branch para *establecerlo* dentro de la branch 4.2, con los siguientes comandos:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a viene a ser la ultima revisión en el histórico de la branch característica antes de que sea bifurcaba de la versión 4.1. Usted puede mirar el histórico de su repositorio git con el comando `git log` para encontrar este.)

Al llegar a este punto, la historia de la branch característica ha sido actualizada, pero no ha sido de hecho fusionada con la versión 4.2 aún. Este le permite a usted resolver conflictos antes de que usted lo fusione a la branch release 4.2. Hacerlo ahora así con los siguientes comandos:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Los branches y los forks y hacer revisiones directamente - ¡Por Dios! Plone uso un repositorio svn, así que todo el mundo es familiar y acostumbrado a hacer revisiones directamente a las branches. Después de la migración de los repositorios svn a los repositorios git en el servicio github, la comunidad decidió mantener este espíritu. Si usted ha firmado el documento contributor agreement, usted puede hacer commit directamente a la branch (para plone esto sería la versión del branch, para más otros paquetes esto sería el branch llamado `master`).

AUN ASÍ, hay unas cuantas situaciones donde una hacer un nuevo branch es apropiado. Si usted:

- usted se esta iniciando,
- usted no esta seguro acerca de sus cambios
- quiere una revisión de comentario/código
- están llevando a cabo un cambio no trivial

Entonces probablemente quieres crear una branch de cualquier paquete que está usando y entonces use la característica de pull request del servicio github para obtener revisión. Todo acerca de este proceso sería el mismo, excepto que necesita para trabajar en una branch. Tome de ejemplo el paquete `plone.app.caching`. Después de comprobarlo con `mr.developer`, cree su propia branch con los siguientes comandos:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Hacer un branch o fork es su elección. Yo prefiero hacer branch, y yo estoy escribiendo la documentación en esto usando el método de branch. Si usted hace un branch, nos ayuda porque nosotros *sabemos* que tienes permisos para aplicar revisiones a este branch. De cualquier forma, es tu decisión.

Proceda como le sea costumbre. Cuándo usted este a punto para hacer `push` de la corrección de su fallo, debe hacer un push a una branch remota con el siguiente comando:

```
> git push origin my_descriptive_branch_name
```

Esto hará un branch remoto en el servicio github. Vaya a esta branch de la interfaz de usuario github y en la parte superior derecha habrá un botón que dice **"Pull Request"**. Este le permitirá hacer una solicitud dentro de un pull request en la branch principal. Hay personas que se ven una vez a la semana o más para revisar las solicitudes pull requests y confirmar si son o no es una buena corrección y le dará una retroalimentación cuando sea necesario. Los revisores son informales y muy agradables, así que no se preocupe - ¡que están ahí para ayudar! Si usted quieres retroalimentación inmediata, valla a la sala IRC con el enlace de `pull request` y pedida una revisión.

Note: ¡todavía necesitas actualizar el archivo `checkouts.cfg` en las branches correctas de proyecto buildout.coredev!

Jenkins ¡Usted TODAVÍA no está listo! Por favor, compruebe que el servicio jenkins se asegure que sus cambios no hallan roto cosas. Se ejecuta cada media hora y tarda un rato para ejecutar la comprobación en una hora es bueno para verificar el resultado que arroje. Ten una cerveza y tu mirada sobre el panel de control Jenkins.

Finalizando Tickets Si usted esta trabajando de un ticket asignado, por favor no olvide en volver a actualizar el ticket y agregar un enlace a sus revisión de cambios. Actualmente no tenemos una integración de nuestro sistema de ticket con el servicio github pero es una forma agradable de seguir sus cambios. Eso también le permite al reportero saber que usted preocupa. Si el fallo es realmente grave, considere en contactar al release manager y he invitarle a hacer un pronto lanzamiento.

FAQ

- ***¿Cómo puedo saber si se tomaron mis cambios de mi paquete?*** Usted puede seguir el proyecto en github y mirar la linea del tiempo de cambios. Usted también puede descargar el `CHANGES.txt` de cada liberación de Plone para ver una lista comprensible de todos los cambios y validar que su contribuciones estén presente.

Escribiendo documentación

Documentación de Plone Como una comunidad, Plone mantienen muchos tipos de documentación:

- Documentos *mantenidos*. Esto es un conjunto limitado de documentación que está pretendido para ser cuidadosamente dirigido y regularmente actualizado.
  - Manual de Usuario.
  - Instalando Plone, la versión original de esta documentación en Ingles esta disponible como Installing Plone.
  - Referencia de Temas
  - Manual de Desarrollo

  Las mejoras a los documentos mantenidos pueden ser discutidos en la lista de correo plone-docs.

- Documentos *editado colectivamente en comunidad*. Estos son abiertos para contribuciones por cualquier que lo deseo hacer. Esto dirige a una riqueza de información que es de más ampliamente variando calidad.
  - Base de conocimiento en plone.org. Cualquiera con una cuenta en plone.org es libre de editarlo.
  - Documentación para el programador en Plone Collective. Cualquiera tal ves puede contribuir.

Documentando un paquete

Lo básico Al menos, su paquete tendría que incluir las siguientes formas de documentación

`README.rst` El archivo README es el primer punto de partida para la mayoría de personas a su paquete. Este será incluido en la página PyPI en su paquete Egg, y en la página de su repositorio github. Este tendrá que ser formateado utilizando reStructuredText (reST) para conseguir el formato apropiado por aquellos sistemas.

`README.rst` tendría que incluir:

- Una breve descripción del propósito del paquete
- Información de instalación (Cómo consigo que funcione?)
- Información de compatibilidad (qué versiones de Plone trabaja con que?)
- Enlaces a otras fuentes de documentación
- Enlaces para reportar errores, listas de correo, y otras maneras de conseguir ayuda.

El manual (a.k.a. documentación narrativa)

El manual va a una profundidad más lejana para personas que quieren saber todo acerca de cómo para utilizar el paquete.

Este incluye temas como los siguientes:

- ¿Qué características poseen?
- Documentos ¿Cómo usar? (¡doctests que no estén en Inglés!)
- Información aproximadamente de la arquitectura
- Común gotchas

El manual debería considerar varias audiencias quiénes pueden necesitar tipos diferentes de información

- Los usuarios finales quiénes utilizan Plone para editar contenido pero no administran el sitio.
- Los administradores del sitio quiénes instalan y configurar el paquete.
- Los integradores quiénes necesitan extender la funcionalidad del paquete desde el código fuente.
- Los administradores de sistemas quiénes necesitan mantener que el servidor ejecute el software.

Los paquetes sencillos con funcionalidad limitada pueden ser una página sola de narrativa documentación. En este caso es más sencillo de incluir él un extendido `README.rst`. Algunos ejemplos excelentes de una página sencilla de archivo readme son http://pypi.python.org/pypi/plone.outputfilters y https://github.com/plone/plone.app.caching

Si su proyecto es moderadamente complejo, puedes querer instalar su documentación con páginas múltiples. La manera mejor de hacer este es para añadir Sphinx a su proyecto y hospedar

su documentos en readthedocs.org de modo que reconstruye la documentación siempre que aplica un cambio en github.com Si tú hace esto, su `README.rst` debe que enlazar fuera sitio a la documentación.

Referencia (a.k.a. documentación de API)

Una referencia de API proporciona información sobre la API pública del paquete (eso es, el código que el paquete expone para uso de código externo.) Eso significa para el acceso aleatorio a recordar el lector de cómo una clase particular o un método trabajo, más que para leer enteramente.

Si el código base está escrito con docstrings, la documentación de la API puede ser automáticamente generada utilizando Sphinx.

`CHANGES.txt` El changelog o registro de cambios es un registro de todos los cambios que hizo al paquete y quién les hicieron, con los cambios más recientes en la parte superior. Esto está mantenido por separado del historial de revisión del git para dar una posibilidad de mensajes mas amigables al usuario al registrar cuándo las liberaciones estuvieran listas.

Un changelog se parece a algo así:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

Ver https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst para un ejemplo completo.

Si un cambio sucedió relacionado a un error en el sistema de ticket, en la entrada changelog tendría que incluir un enlace a aquel asunto.

Licencias Información sobre la licencia del código abierto utilizada para el paquete tendría que ser colocado dentro del directorio `docs`.

Para paquetes del núcleo Plone, esto incluye `LICENSE.txt` y `LICENSE.GPL`.

Utilizando Sphinx

Referencias reST:

- Documentación Shpinx orientada a Plone
- Manual básico Sphinx reST

Guía de estilos Python, como cualquier lenguaje de programación, puede ser escrito en un número de estilos. ¡Somos los primeros en admitir que Zope y Plone no son los mejores ejemplos de la integridad estilística, pero eso no nos impide intentarlo!

Si no estas familiarizado con PEP 8 - la guía de estilo de la python, por favor, tome un momento para leer y ponerse al día. Nosotros no requerimos, pero nosotros, como comunidad realmente, realmente lo apreciamos.

Convenciones de nombre ¡Por encima de todo, ser coherente con cualquier código que usted está modificando! Históricamente, el código es todo camel case, pero muchas nuevas bibliotecas se encuentran en la convención PEP8.

La lista de correo es la explosión de debates sobre que es lo mejor, así que nosotros vamos a dejar el ejercicio de decidir qué hacer con el usuario de esta lista.

Convenciones de Archivo En Zope 2, los nombres de archivos que solía ser así MixedCase. En Python, y así en Plone yendo de frente, preferimos todos los nombres de archivos en minúsculas. Esto tiene la ventaja de que usted puede ver inmediatamente si se refiere a un módulo / archivo o una clase:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

comparar eso a:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Los nombres de archivos debe ser ordenados y descriptivos. Piensa acerca de como un import podría leerse:

```
from Products.CMFPlone.utils import safe_hasattr
```

comparar eso a:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

A la primera es, obviamente, mucho más fácil de leer, menos redundante y por lo general estéticamente más agradable.

Note: Este ejemplo es casi tan terrible como el que viene. Necesitamos una mejor.

Reglas específicas

- ¡No utiliza tabuladores en código de Python! Use espacios como indentado, a 4 espacios para cada nivel. Nosotros no **"requerimos"** PEP8, pero mucha gente lo usa y eso es bueno para usted.
- Indentar apropiadamente, incluso en HTML.
- Nunca usar una excepción bare. Cualquier cosa como `except: pass` es probable que sea revertido instantáneamente
- Evitar `tal:on-error`, desde estas excepciones swallows
- No use `hasattr()` - esto son swallows exceptions, por favor, en este caso use `getattr(foo, 'bar', None)`. El problema con las swallowed exceptions,no es sólo el informe de errores pobre. Esto también puede enmascarar conflictos de errores (`ConflictErrors`), que indican que algo ha ido mal en el nivel de la ZODB!
- Nunca, coloque ningún código HTML dentro del código Python y lo retorne como una cadena
- No adquirir nada menos que sea absolutamente necesario, especialmente herramientas. Por ejemplo, en vez de utilizar `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- No coloque mucho de la lógica en las Plantillas de pagina Zope - ZPT (use Views en este caso!)
- Recuerde agregar etiquetas i18n en ZPTs y código Python

Implementando PLIPS

Todo sobre los PLIPS

¿Qué es un PLIP? Del ingles, PLIP significa PLone Improvement Proposal, el cual es una propuesta de mejora a Plone. Eso es cuando un cambio a un paquete Plone que afectaría todo el mundo Las PLIPs van por un proceso diferente que la corrección de fallos debido a su ancho logrando efecto. El equipo del framework Plone 4.x revisa todas las PLIPs para ser seguro que es en el mejor interés de la comunidad para ser implementada y que es de calidad alta.

¿Es un PLIP o una corrección de fallo? En general, cualquier cosa que cambia el API de Plone en el backend o la UI en la portada tendría que ser archivado como PLIP. En caso de duda, se presentará como una PLIP. El equipo del framework esta ansioso al reducir es propia carga de trabajo y clasificarla de nuevo por usted.

¿Quiénes pueden presentar PLIPs? Cualquiera quién ha firmado el Plone core contributor agreement puede trabajar en un PLIP. No deje que la redacción le asuste: la firma del acuerdo es fácil y usted tendrá acceso casi inmediato. Usted no tiene que ser el más increíble programador en el mundo entero para poder presentar una PLIP. El equipo del Framework estará encantado de ayudarle en cualquier momento del proceso. La presentación de un PLIP puede ser un gran proceso de aprendizaje y animamos a la gente de todos los orígenes a presentar. Cuando el PLIP es aceptado, un miembro del equipo del Framework "defenderá" su PLIP y a ser encargado de llevar a cabo esta mejorar hasta que este completada. Las PLIPs no solamente son código fuente monkeys. Si usted tiene ideas sobre nuevas formas de interactuar o de la interfaz de usuario sus ideas son mas que bienvenidas. Incluso te ayudaremos a la par con los implementores si necesario.

¿Qué es un defensor PLIP? Cuando usted presenta su PLIP y esta es aprobada, Un miembro del equipo de Framework que este especialmente animado a ver hecho realidad el PLIP sera asignado a su PLIP como un defensor. Ellos están ahí para animar hasta su culminación, así que responde a cualquier pregunta y proporcionar orientación necesaria.

Un defensor debería:

- Responda a las preguntas formuladas por el implementador tiene sobre el PLIP, técnica y de cualquier otra
- Fomentar el autor PLIP por su constantemente retroalimentación y estímulo
- Cuidar que el implementador logre la lineas de tiempo y coloque las cosas listas a tiempo
- Asistir en la búsqueda de ayuda adicional cuando sea necesario para completar la implementación en un tiempo oportuno

Ten en mente que los defensores están en modo pasivo por defecto. Si usted necesitas ayuda o asesoría, por favor, contacte a ellos lo mas pronto sea posible para activar el modo ayuda.

Yo aun estoy nervioso. ¿Puedo yo involucrarme de otras formas primero? Si quieres sentir el proceso y cómo funciona, nos ayudan a revisar los PLIPs terminar las implementaciones. Si usted quiere sentir el proceso y cómo funciona, ayúdenos a revisar los PLIPs para terminar las implementaciones. Simplemente pregunte a los miembros del equipo de Framework cual PLIPs esta disponible para revisar o verificar el estatus del PLIPs en el siguiente enlace. Asegúrese de decirnos como usted intenta revisar el PLIP uniéndose a la lista de correo del equipo de Framework y enviar un correo rápidamente. Entonces, siga las simples instrucciones para revisar un PLIP. ¡Gracias de antemano!

¿Cuándo yo puedo presentar un PLIP? ¡Hoy, mañana, en cualquier momento! Después que la PLIP es aceptada, el equipo del Framework tratara de juzgar la complejidad y el tiempo para completar y asignar este en un hito. Usted puede iniciar rápidamente su trabajo, y nosotros animamos a presentarlo 'rápido y furioso'.

¿Cuándo el PLIP es debido? Resumen: Tan pronto usted lo finalice. Técnicamente, nosotros queremos ver eso completado para la publicación a la cual fue asignada. Nosotros sabemos que todos tenemos ocupaciones que nos ponen muy ocupados y además de nuevos problemas hacen a los PLIPs más complicado si es así entonces los vamos a mover hacia la siguiente versión. En general, nosotros no queremos seguir un PLIP por mas de un año. Si su PLIP es aceptada y nosotros no vemos actividad en mas de un año, nosotros probablemente le preguntaremos para iniciar de nuevo el proceso.

No le gusta mi PLIP :(Y ahora que? Solo por que un PLIP no ha sido aceptado en núcleo no significa que sea una mala idea. A menudo, es el caso que hay implementaciones compitiendo y nosotros queremos ver eso evaluado como un añadido antes de recibir la "bendición" la implementación preferida.

Información general del proceso

1. Presentar un PLIP (en cualquier tiempo)
2. PLIP es aprobada para incluirse dentro de núcleo para ser dado en una publicación
3. El desarrollador implementa el PLIP (código, tests, documentación)
4. PLIP es presentado para ser revisado por un desarrollador
5. El equipo del Framework revisa el plip y ofrece un comentario
6. El desarrollador direcciona lo concerniente en al comentario y presenta de nuevo si es necesario. Esto puede ir y venir varias veces hasta que tanto el FWT y el desarrollador está satisfecho con el resultado.
7. El PLIP es aprobada para fusionarse. En raras circunstancias, un PLIP sera rechazado. Esto es usualmente a que el resultado del trabajo hecho por el desarrollador no responde al comentario o abandono del proceso. ¡Aguante ahí!
8. Después de que todos los otros PLIPS son fusionado, una publicación se corta. Esperar por los nuevos fallos!

Como presentar a PLIP

> Si usted necesita ayuda en algún punto de este proceso, por favor, contacte a un miembro del equipo del framework personalmente o pida ayuda en la lista de correo FWT.

Un PLIP es un ticket con una plantilla especial. Para iniciar, abra un nuevo ticket y seleccione "PLIP" como el tipo de ticket. Un nueva plantilla de ticket se cargara y usted debería planear llenarla en todas sus secciones y campos.

Cuando escribe una PLIP, de se lo mas especifico que pueda al punto a tratar. Recuerde su audiencia - que le puede dar apoyo a su propuesta, las personas tendrán que ser capaces de leerla! Un buen PLIP es suficientemente claro para un usuario conocedor de Plone este disponible a entender la propuesta de cambios, y suficientemente detallada para que el release manager y otros desarrolladores entiendan el completo impacto de la propuesta puede tener el código base. Usted no tiene que listar cada línea de código que necesidades de ser cambiadas, pero usted también debe dar una indicación que usted tenga sobre alguna idea de como el cambio puede ser viablemente implementado.

Si el cambio es menor entonces un ticket en el sistema de seguimiento debe ser suficiente, añadido como una mejora. El punto clave acá es que cada cambio necesita documentación para que otros usuarios puedan ver que cambios hay de nuevo. Esto puede ser en el formulario de un registro de en el sistema de incidencia, o un PLIP en el caso de que es un cambio mayor. Un fallo o cambio menor normalmente no necesita ir a través del proceso de revisión - que hace un PLIP.

Después que su plip es escrita, solicite un comentario de sus ideas en la lista de correo plone-developers. En este proceso de evaluación, usted quiere asegurarse de que el cambio no afectará negativamente a otras personas en un accidente. Otros pueden ser capaces de señalar puntos de riegos o incluso ofrecer una mejor o soluciones existentes.

Cuándo este feliz con el comentario, *presente un PLIP*. Por favor, use la plantilla proveída (XXX: put the template here? Can we just have a custom ticket type?). Por favor, note un par de cosas. Es muy rara ves que la sección de "Risks" estará vacío o nula. Si usted busca esto es el caso y su PLIP no tiene más nada que un cambio trivial, quizás con algo más investigación debería estar listo.

El campo seconder es REQUERIDO. Nosotros enviaremos el PLIP de regreso a usted si ese campo no es llenado. En la actualidad, sólo alguien más que piensa que su PLIP es una buena idea, un +1. En el futuro cercano, vamos a empezar a preguntar al secunde que es un socio de la codificación, o alguien que esté dispuesto y sea capaz de terminar el PLIP si algo le ocurriera al implementador.

Todo lo demás debe explicarse por sí mismo en el ticket de su PLIP. Si le dio pereza escribir estas PLIP tan documentado. Estoy apostando a este último.

Evaluando PLIPs Después de presentar su PLIP, el equipo Framework tendrá un par de semanas para conocer y le harán de conocimiento si el PLIP es aceptado. Si el PLIP no es aceptado, por favor, se sienta mal! Nosotros animamos a que la mayoría de los PLIPs pasen a través del proceso de add on al principio, si es posible para asegurarse de que la mayoría de la comunidad lo use.

Toda la comunicación con usted ocurrirá en ticket PLIP ticket en si mismo por favor, este pendiente a su bandeja de entrada de su correo electrónico por cada cambios aplicado al ticket.

Estos son los criterios por del equipo framework que va a revisar en su revisión del paquete:

- ¿Cual es el tamaño y estatus del trabajo necesitado para culminarlo? ¿Esta listo un add-on y esta bien establecido?
- ¿Esta idea esta bien cocinada y se expresa con claridad?
- ¿El trabajo propuesto en Plone deber estar ahora, en el futuro?
- ¿Es esta PLIP más apropiado a ser calificado como un add-on?
- ¿Es esta PLIP demasiado arriesgada?

Vea la pagina Revisando PLIPs para más información.

Implementando su PLIP Usted puede iniciar el desarrollo en cualquier momento - pero si usted yendo a modificar el Plone en si mismo, usted podría querer esperar para ver si sus ideas son aprobadas primero para salvarse algún trabajo que no sea necesario.

Reglas Generales

- Cualquiera de los nuevos paquetes debe estar en una branch bajo el namespace plone en github. Así usted no tenga el desarrollo allí, pero ese debe estar allí cuando este presentado. Nosotros recomendamos usar branches en el repositorio github.com/plone y se detallará abajo.
- Lo mas importante, los revisores PLIP debe estar disponible a ejecutarse con buildout y cada cosa debería "solo trabajar" (tm).
- **Cualquier nuevo código fuente debería:**
  - Estar documentado apropiadamente
  - Tener el código fuente limpio
  - Usa los idiomas actuales de desarrollo
  - Estar probado

Creando una nueva branch PLIP Cree un archivo de configuración buildout para su PLIP en la carpeta llamada `plips`. Dar un nombre descriptivo, iniciando con el numero de plip; por ejemplo `plip-1234-widget-frobbing.cfg`. Es archivo sera definido en el branches/trunks que usted esta trabajando con su PLIP. Esto debería lucir algo así como lo siguiente:

En el archivo `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
```

```
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Utilice la misma convención de nombre cuando haga un branch existente a paquetes existentes, y siempre usted debería siempre hacer branch paquetes cuando esta trabajando en las PLIPs.

Finalizando Antes de hacer que su PLIP este listo para revisar, por favor añada un archivo dando un conjunto de instrucciones al revisor PLIP.

Este debería ser llamado `plip_<number>_notes.txt`. Este debe incluir (pero no es limitado a solo eso):

- Las direcciones URLs que apuntan a toda la documentación creada/actualizada
- Cualquier inquietud, las incidencias aún permanecen
- Cualquier cosas buildout extrañas
- XXX: ¿Algo mas?

Una ves que allá finalizado, por favor, actualice su ticket indicar que esta listo para ser revisado. El equipo del Framework asignara 2 a 3 personas para revisar su PLIP. Ellos seguirán las lineas guías definidas en Revisando PLIPs.

Después de que el PLIP ha sido aceptado por el equipo de framework y el release manager, se le pedirá que fusionar su trabajo dentro de la línea de desarrollo principal. La fusión de la PLIP no es la parte más difícil, pero hay que pensar en ello cuando se desarrolla. Usted tendrá que interactuar con un gran número de personas para conseguir que todo quede preparado. La fusión puede causar problemas con otros PLIPs que vienen en camino. Durante la fase de fusión debe estar preparado para ayudar con todas las características y fallos que puedan surgir.

Si todo ha ido según lo previsto en el próximo lanzamiento de Plone contendrá su PLIP en ella. Se espera contar con su ayuda para dar soporte a esa característica después de haber sido lanzado en Plone (dentro de lo razonable).

Troubleshooting: solucionar problemas

Incidencias Buildout Buildout puede ser muy frustrante para aquellos no familiarizados con el análisis a través del lenguaje robot autista. ¡No tema! Estos errores son casi siempre tienen una solución rápida y un con poco de comprensión le lleva a un largo camino.

Errores ejecutando bootstrap.py Ni siquiera se puede llegar a correr buildout y entonces ya tendrá un error. Déjeme darle un ejemplo de esto:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

Usted puede pensar que la diosa buildout está enfadada porque han pasado MESES desde que usted hizo un sacrificio humano para ella, pero sea fuerte y siga adelante. Buildout simplemente se ha dado cuenta de que la versión de buildout requerida por el archivo bootstrap.py que usted está intentando ejecutar no coincide con la versión de buildout en la biblioteca de python. En el error anterior, el sistema dispone de buildout 1.5.1 instalado y el archivo bootstrap.py quiere correr con 1.5.2.

Para solucionarlo, hay un par de opciones. En primer lugar, puede hacer que se ejecute buildout con la versión que ya se han instalado mediante la invocación de la etiqueta de versión. Esto le dice a su archivo bootstrap.py [Plone] para ejecutar muy bien con la versión que ya se ha instalado. En el caso de que el error pegado encima, que sería:

```
> python bootstrap.py --version=1.5.1
```

Yo conozco personalmente que las versiones 1.4.4, 1.5.1, 1.5.2 y todo funciona de esta manera.

La otra opción es eliminar el paquete egg actual y forzar la actualización. En el caso de que el error anterior, todo lo que necesita para eliminar del paquete egg que tiene actualmente en ell sistema. ej:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

Al volver a ejecutar el archivo bootstrap.py, este buscará para el paquete egg para el buildout, ya que no hay uno, y luego ir a buscar el paquete egg con la nueva versión que se quiere para su construcción buildout.

De una de esas, digamos dos Ave Marías, con tres padres nuestros y ejecute de nuevo bootstrap.py. ¡Tada!

Otra cosa de la nota es que está ejecutando bootstrap efectivamente asocia al ejecutable python y todas sus bibliotecas a su buildout. Si tiene varias instalaciones Python y quiere cambiar a cual Python está asociando a su buildout, sólo tiene que volver a ejecutar bootstrap.py con el nuevo python (y vuelva a ejecutar el buildout). Usted puede obtener el mismo error anterior nuevamente, ahora que sabes cómo solucionarlo, usted puede pasar ese tiempo bebiendo cerveza en vez de aplastar el teclado.

¡Hurra!

Cuando Mr. Developer no es feliz `mr.developer` nunca es feliz, excepto cuando es. A pesar de que esto técnicamente no es un incidente buildout, pasa cuándo esta ejecutando buildout, entonces yo lo estoy poniéndolo en las incidencias buildout.

Cuándo trabajando con la instancia de desarrollo, especialmente con toda el movimiento de ida y vuelta de cambios entre github y svn, usted puedes tener una copia vieja de un paquete en le directorio `src`. El error luce así:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

Mientras usted no tenga cualquier revisión de cambios pendiente, usted solo necesita remover el paquete del directorio `src/` y se vuelve a revisar para usted cuando se actualiza.

Usted también se puede conseguir con errores tan divertidos tales como:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

Estos errores están bien para ser ignorados SI y SÓLO SI las líneas que siguen diga:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

Si buildout termina con un aviso de que algunos paquetes no se pudo descargar, entonces es probable que el paquete no se ha descargado. Esto es malo y puede causar todo tipo de errores al iniciar o tratar de hacer las cosas porque nunca se llegan a descargar el paquete.

Hay dos maneras de obtener este error desaparezca. Lo primero es eliminar todas las instancias del host filtrado. registrar minuciosamente todos los archivos y elimine cualquier lineas la cual diga así `allow-hosts =` y `allow-hosts +=`. En teoría, mediante la restricción de los hosts del cual se va a descarga, buildout irá más rápido. Ya sea que realmente sucede o no, yo no puedo opinar. El punto es que son de seguramente se pueden eliminar.

La segunda opción es el permitir al host al cual ese apunta para ser añadido de algo como esto a su archivo .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Otra vez, esto es sólo necesario si el paquete no fue encontrado al final.

¡Hurra!

Errores de ruta mr.developer `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

Cuando ejecuta algún comando **`./bin/develop`**.

Para solucionar, simplemente ejecute el siguiente comando:

```
ln -s plips/.mr.developer.cfg
```

Otros incidentes aleatorios

Paquetes Sucios "ERROR: Can't update package '[Some package]', because it's dirty."

Solución `mr.developer` se queja porque un archivo se ha cambiado / añadido, pero no se ha generado una revisión.

Utilice el comando **`bin/develop update --force`**. Añadiendo `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` a su configuración subversion global-ignores ha sido sugerido como una solución más permanente.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Parece no ser en realidad el caso. Eliminar el archivo `mkzopeinstance.py` desde el directorio `bin/` y ejecute de nuevo el script buildout para corregir esto si usted está encontrando fastidioso.

No puede abrir el archivo '/Startup/run.py' Dos posibles soluciones, usted esta usando Python 2.4 por equivocación, así que por favor use Python 2.6 en cambio. O, usted tal ves necesitar asegurarse que usted ejecuto el script **`bin/buildout ...`** después ejecutar el script **`bin/develop ...`**. Trate de remover los directorios `parts/*`, `bin/*`, el archivo `.installed.cfg`, entonces ejecute de nuevo el archivo `bootstrap.py` y ejecute de nuevo buildout, develop, buildout.

PIL perdido El archivo `pil.cfg` es incluido con esta configuración buildout para ayudar en instalación PIL. Ejecutar **`bin/buildout -c pil.cfg`** a instalar la librería PIL. Este método no funciona en Windows, así que nosotros somos incapaces de correr él por defecto.

Incidencias con paquetes egg modificado El comando **`bin/develop status`** el mostrara que el paquete egg `Products.CMFActionIcons` ha sido modificado, pero no lo he tocado. Y ejecutando este comando **`bin/develop up`** esta previniendo la actualización de todos los paquetes egg.

Solución Editar el archivo `~/.subversion/config` y añadir eggtest*.egg a la lista de `global-ignores`

Cómo actualizar estas documentaciones Estos documentos actualmente son almacenados con el directorio `/docs`. Para actualizarles, por favor, hacer un checkout desde el buildout coredev y actualizar allá. Hacer los cambios en la rama de versión más reciente (como de esta `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

Para probar sus cambios localmente, inicie de nuevo el buildout y entonces:

```
> bin/sphinx-build docs docs/build
```

Sphinx colocara en un directorio que usted puede consultar en tu navegador web para validar. Por ejemplo: `file:///home/user/buildout.coredev/docs/build/index.html`

Por favor asegúrese en validar todos los avisos y los errores antes de generar una revisión para estar seguro que los documentos queden válido. Una vez todo está a listo para ir, haga commit y push de sus cambios.

El comando `git cherry-pick` hace commits en la rama más reciente a la rama actualmente liberada (como de esta escritura `4.3`) si estos cambios aplican a aquella versión (usted puede conseguir el SHA hash de registro **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

Tal ves allá conflictos; entonces, resuélvalos y seguir las instrucciones que la herramienta git le da a usted para completar el comando **`git cherry-pick`**.

Otros Estos son algunos documentos a utilizar como referencia para esta documentación.

Acuerdo de contribución para Plone explicado Para los colaboradores potenciales al código base del núcleo de Plone les es requerido que firmen el contributor's agreement, el cual asigna copyright del código fuente a la Fundación Plone, la organización sin ánimo de lucro qué administradores auxiliares el código base de Plone. Este documento explica el propósito de este respondiendo a este con preguntas y respuestas acerca del lo que este significa.

El acuerdo de contribución para Plone puede ser encontrado en: http://plone.org/foundation/contributors-agreement/agreement.pdf

Acerca del acuerdo de contribuidor de Plone La Fundación siente que beneficia la comunidad teniendo una organización sola para mantener los derechos a Plone. Antes de la Fundación, la propiedad intelectual de Plone era conjuntamente mantenida por desarrolladores individuales y por Alan Runyan y Alexander Limi. Los miembros comunitarios quiénes formaron la Fundación sentían que habiendo eso, la Fundación cuidaría esos derechos y que proporcionaría varios beneficios:

1. **Minimizando la confusión / maximizando la compatibilidad empresarial** – Para las organizaciones que consideran la adopción Plone tener una respuesta sencilla para "¿Quien es el dueño de esto?", en lugar de una respuesta más complicada que podría ahuyentar el carácter cauteloso.
2. **Protección de la marca** – Por que teniendo a la Fundación para proteger la marcas comerciales y derechos de marca sobre los activos Plone, puede proteger eficazmente estos ante el uso injusto.
3. **Garantía de versiones futuro en código abierto** – El contributor agreement de la Fundación asegura que habrá **siempre** de ser una versión aprobada por la Open Source Initiative OSI de Plone.

Preguntas & Respuestas ¿Qué cubre el contributor's agreement?

Este contributor's agreement es sólo para el código base del núcleo de base de Plone. El código base del núcleo de base de Plone es el código que se vive en los repositorios de la versión del núcleo de Plone, actualmente ubicada en http://dev.plone.org/plone y https://github.com/plone. Las contribuciones a al repositorio "Collective," actualmente localizado en http://dev.plone.org/collective y https://github.com/collective no están asignados a la Fundación Plone y se ponen a disposición bajo cualquier licencia de los desarrolladores del proyecto desean utilizar, aunque los productos add-on que se importar como código Plone basado en GPL, por supuesto, sujeto a los términos de la licencia GPL, que obliga a los trabajos derivados a ser licencia GPL.

¿Qué derechos voy a seguir teniendo por mis contribuciones?

A los colaboradores se les pide a transferir sus derechos de propiedad intelectual a la Fundación. A cambio, se les dará de nuevo los derechos irrevocables para usar y distribuir sus contribuciones. ncluso pueden dar a sus contribuciones a otros proyectos de código abierto (siempre y cuando esos proyectos sean compatibles con la propia licencia con que está emitida Plone) o usar en aplicaciones comerciales de fuentes no abiertas (si esta es compatible con la propia licencia con que está emitida Plone).

¿Tengo que firmar el contributor's agreement para hacer checkins para el código base del núcleo de Plone?
Si.

¿Tengo que firmar el contributor's agreement para suministrar un parche para el código base del núcleo de Plone?

Acogemos con entusiasmo parches, pero no podemos hacer el merge hasta que firme y devuelva el contributor's agreement. (A menos que, a juicio del Plone Release Manager, el parche es tan pequeño que no constituyan un "trabajo creativo". Véase la Política de Acuerdos del contribuyente y parches para más detalles sobre esta política.)

¿Puedo conceder a la fundación Plone una licencia no exclusiva para mis contribuciones en lugar de una licencia exclusiva, por lo que yo puedo aportar el mismo código en otros proyectos bajo diferentes términos o usar la contribución de otras actividades comerciales?

No bajo la versión actual del contributors agreement.

¿La Fundación Plone hace control del uso de la marca?

Si. A fin de mantener la marca, la Fundación (o cualquier propietario de la marca) deben demostrar que han tomado medidas para protegerlo.

Plone estará siempre disponible bajo una licencia aprobada por la Open Source Initiative/Código abierto? ¿No podría el Board cambiar la opinión sobre esto?

Plone siempre estará disponible bajo una licencia aprobada por la Open Source Initiative, lo que está escrito en el lenguaje del contributor agreement de cada desarrollador y la firma de la fundación.

¿Plone nunca estará disponible bajo una licencia no compatible con la GPL?

El enfoque actual Plone establece que las empresas pueden negociar una licencia no-GPL. De este modo, la Fundación podría perseguir un esquema de licenciamiento dual (GPL y no GPL) -, pero, en este momento, el Board aún no ha creado ninguna política al respecto. Esta es una pregunta importante para la comunidad, por supuesto, y la Fundación tiene la intención de tener esta conversación de una manera transparente.

¿Por qué iba alguien a querer un licenciamiento no GPL para Plone?

Dos posibles razones: Algunas compañías se resisten a hacer modificaciones internas de los sistemas de framework (como Plone) que están bajo la GPL, por temor a que una cláusula de la licencia GPL podría obligarlos a revelar su trabajo interno - queriendo así a licenciar en los términos de (por ejemplo) una licencia estilo BSD. En segundo lugar, las empresas pueden ofrecer una versión comercial de Plone, bajo una licencia convencional cerrada, sin la obligación de revelar el código fuente o de compartir los cambios.

¿Cuánto costaría una versión no GPL de Plone?

> ¿Una pequeña empresa será capaz de pagar uno? – Ni la Fundación ni el Board han tomado ninguna decisión acerca de una versión no GPL, y mucho menos acerca de los precios. Sin embargo, uno de los objetivos declarados de la Fundación es la de mantener la igualdad de condiciones para Plone al tratar de beneficiar a todos los bienes comunes de Plone. Si una versión no GPL estaba disponible, y una gran empresa la compró, añade características a la misma, y lo vendió, ¿no se utilizará nuestro trabajo sin una obligación de devolver? Es útil recordar el valor de la base de código abierto ofrece: desarrollo distribuido, mantenimiento, control de seguridad y apoyo. Las empresas que construyen grandes características para Plone **ya están** teniendo que tomar decisiones acerca de si para liberar sus productos bajo una licencia de código abierto o no (ya que siempre podrían ponerlos en libertad como un producto y no como una modificación al núcleo de Plone). A pesar de esto, sin embargo, muchas contribuciones amplias e inmejorables - como Archetypes - se han hecho, y la Fundación espera que las empresas seguirán haciéndolo. En cualquier caso, una empresa que compra una licencia no GPL (debe tal vez estar haciendo disponibles) contribuyendo con recursos financieros a nuestra comunidad, que puede estar usado para desarrollar, comercializar y proteger a la versión GPL de Plone.

Mr. Developer Esta configuración buildout usa mr.developer para administrar los paquetes de desarrollo. Ver http://pypi.python.org/pypi/mr.developer para mayor información o ejecutar el comando **`bin/develop help`** para un lista de comando disponibles.

El mas común flujo de trabajo para obtener todas la ultimas actualizaciones es:

```
> git pull
> bin/develop rb
```

Esto te conseguirá la versión mas reciente de la configuración **coredev**, compruebe y actualice todos los paquetes vía Subversion en el directorio `src` y ejecute buildout para configurar el asunto.

De vez en cuando puedes comprobar si alguna cosas pendiente:

```
> bin/develop st
```

Si esto imprime alguna líneas con un signo de interrogación delante, usted puede limpiarlo con:

```
> bin/develop purge
```

Esto sacará los paquetes del directorio `src/` los cuáles ya no son necesarios, cuando estos han sido reemplazados por una apropiada liberación en formato Egg de estos paquetes.

Revisando PLIPs

Expectativas Una buena revisión del PLIP toma aproximadamente 4 horas así que por favor haga planes en consecuencia. Cuando haya terminado, si usted tiene acceso al núcleo por favor comprometerse con la revisión de los plips y hacer referencia a la PLIP en el mensaje de confirmación. Si no tienes acceso, por favor adjuntar su revisión al ticket PLIP en si mismo.

Instalando el entorno Seguir las instrucciones en definiendo un entorno de desarrollo para "obtener el código fuente". Usted necesitará a tener una copia al branch del repositorio del cual el PLIP fue asignado. En vez de correr el buildout con el archivo predeterminado buildout, usted lo ejecutara el archivo especifico de configuración del plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisión de funcionalidad Hay varias cosas que podría ser dirigido en una revisión PLIP que depende de la naturaleza del PLIP. Esto es de ninguna manera una lista exhaustiva, pero un sitio para empezar. Las cosas para pensar aproximadamente cuándo este revisando:

General

- ¿Este PLIP realmente hace lo que los implementadores propusieron? ¿Hay variaciones incompletas?
- ¿Hubo allí algunos errores al ejecutar el buildout? ¿Hizo el trabajo de migración?
- ¿No mensajes de error y los mensajes de estado tiene sentido? ¿Son correctas las internacionalizaciones?
- ¿Hay alguna consideraciones de rendimiento? ¿El implementador de este PLIP se dirigió a ellos si es así?

Errores

- ¿Hay algún error? Nada es demasiado grande ni pequeño.
- ¿Los campos a manejar data estrafalarias? ¿Cómo son las cadenas en los campos de fecha o los campos nulos son requeridos?
- Es validación y cuestión de puro sentido? ¿Es demasiado restrictiva o no es lo suficiente restrictivo?

Incidentes de usabilidad

- ¿Es usable la implementación?
- ¿Cómo los usuarios finales novatos responden al cambio?
- ¿Hace este PLIP necesidad una revisión de usabilidad? Si usted piensa que este PLIP necesita una revisión de usabilidad, por favor, cambiar el estado a "please review" y agregue una nota a estos comentarios.
- ¿El PLIP es compatible con el resto de Plone? Por ejemplo, si hay configuración de panel del control, hacer esta forma nueva funciona en con el resto de los panel de control.
- ¿Todo se comporta bien para principiantes y usuarios avanzados? ¿Hay algún flujo de trabajo que se ofrece una experiencia extraña?
- ¿Hay permisos nuevos y ellos trabajan correctamente? ¿Esas asignaciones de roles hacen con sentido?

Incidencias de la documentación

- ¿Hay suficiente documentación disponible correspondiente para el usuario final, ya sea programador o usuario plone?
- ¿Este cambio en si mismo esta correctamente documentado?

Por favor, reporte fallos/incidencias en el sistema de ticket Trac como si se tratara de cualquier fallo Plone. Haga referencia al PLIP en el fallo, asigne a su implementador, y añadir una etiqueta para el PLIP en forma de plip-xxx. De este modo el implementor puede encontrar ayuda si lo necesita. Por favor, también priorizar el ticket. El PLIP no será fusionado hasta que todo los fallos bloqueadores y críticos están corregido.

Revisión de Código

Python

- ¿Es este código fuente mantenible?
- ¿Esta el código fuente adecuadamente documentado?
- ¿El código fuente se adhiere al estándar PEP8 (más o menos)?
- ¿Están importando módulos obsoletos?

Javascript

- ¿El javascript conocer a nuestro conjunto de normas javascript? Ver http://developer.plone.org/templates_css_and_javascripts/javascript.html
- ¿El Javascript funciona en todos los navegadores soportados actualmente? ¿Es eficaz?

ME/TAL

- ¿La vista de usuario del PLIP adecuadamente y evitando demasiada lógica?
- ¿Hay algún código fuente en un bucle que podría ser un problema de rendimiento?
- ¿Hay alguna líneas de código con estilo obsoleto o viejo ME/TAL como el uso de DateTime?
- ¿Es compatible con lo que dicta el estándar html? ¿Los ids y class CSS utilizado son apropiados?

Ejemplos de revisiones de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Traducciones disponibles Hay algunas iniciativas para mantener las traducciones disponibles en otros idiomas para esta documentación como las siguientes:

Process for Plone core's development

Introduction This documentation describes the process for developing Plone. It is primarily a technical resource for setting up your core developer buildout, fixing bugs, and writing plips.

STOP! Legally, you can NOT contribute code unless you have signed the contributor agreement. This means that we can NOT accept pull requests from you unless this is done, so please don't put the code reviewers at risk and do it anyways. Submitting the agreement is easy (and will soon be easier) and if you want quick access and are familiar with the community, go into irc and ask one of the repo admins to give you access with a scanned copy of the agreement. They will get you going as fast as possible!

Table of Awesome

Contributing to Plone Core There are many people and companies who rely on Plone on a day-to-day basis so we have to introduce some level of code quality control. Plone's source code is hosted in a git repository at https://github.com/plone, but only members of the developer team have commit-rights.

Just sending in a contributors agreement does not guarantee you access to the repository, but once you send it in we will always have it on file for when you are ready to contribute. We do ask that before requesting core access you familiarize yourself a little with the community since they will help you get ramped up:

- Ask and (especially) answer questions on stack overflow and IRC with a focus on getting to know the active developers a bit.
- Attend a conference / symposium or participate in a sprint / tune-up. There are plenty of opportunities to meet the community and start contributing through various coding sessions, either in person or on the web. You may even be able to get immediate core access at a conference if you are flexing your mad coding skills and the right people are attending.
- Get your feet wet by contributing to the collective. Don't worry about getting it perfect or asking for help; this way you get to know us and we improve our code together as a community.
- **Patches:** Historically we encouraged people to submit patches to the ticket collector. These tickets are usually ignored forever. Technically, in order for us to accept your patch you must sign the contributors agreement. If you want to contribute fixes, please just sign the agreement and go through the standard github pull request process described until you feel comfortable to bypass review. If the ticket is trivial, or you're fixing documentation, you do not need to sign a contributor's agreement.

Once you have familiarized yourself with the community and you are excited to contribute to the core:

- Sign the contributor agreement at http://plone.org/foundation/contributors-agreement/agreement.pdf/at_download/file, then either snail mail it to the address provided or scan and email it to assignments@plone.org. This offers both copyright protection and ensures that the Plone Foundation is able to exercise some control over the codebase, ensuring it is not appropriated for someone's unethical purposes. For questions about why the agreement is required, please see Contributor's Agreement for Plone Explained.

If you aren't sure where to start or just want more direction, feel free to get on IRC, mailing lists, Twitter, etc... and ask for help. While there is no official mentoring process, there are plenty of people willing to act in that role and guide you through the steps of getting involved in the community. A common way to start contributing is to participate in a Plone tune-up day. Tune-ups are filled with a good mix of newbies and experienced devs alike. For more information, please see http://plone.org/tuneup.

Welcome to the Plone community!

Dealing with pull requests on GitHub Before we can merge a pull request, we have to check that the author has signed the contributor's agreement.

If they're listed in https://github.com/plone?tab=members, the author has signed so we can go ahead and merge.

If they aren't listed there, there's still a chance they have signed the contributor's agreement. Check on IRC #plone-framework.

Pull requests without contributor's agreement can only be merged in trivial cases, and only by the release manager.

Plone Developer Culture If you are going to be contributing back to Plone, we ask a couple things. First, please join the plone-developers list and at minimum lurk around. You will quickly see how people work and what kind of things are best suited for group discussion. Second, please ask for help setting up your environment in IRC. Most of our developers work there and you will get the best advice there.

Download an IRC client (Or using an alternative client through the web) and jump on to #plone-framework (and/or #plone - both on freenode). The people in #plone-framework have been using plone for a very long time and are happy to help you get going and make the right decisions. More info on IRC can be found at http://plone.org/support/chat.

If you are actively committing code, join the test bot mailing list so you know if your recent commits have broken (or fixed!) the build.

If you are in a timezone when things are not very active, please post to the plone-developers mailing list or grab a beer and wait for people to wake up.

When in doubt, please ask. The code base is very complicated and everyone is vested in the right thing happening. Despite the occasional grouch here and there, most plone devs will go out of their way to get you on the right path.

Getting started with development This document assumes you want to run the current latest Plone source, fix a bug in Plone, or test an addon in the context of the latest code, and will detail the full process. For more information on writing PLIPS, please go here.

Version Support Policy If you are triaging or fixing bugs, keep in mind that Plone has a version support policy.

Dependencies

- Git
- Subversion
- Python 2.6 or 2.7 including development headers.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Make sure to install this into the proper python environment.
- VirtualEnv in the proper python environment.
- GCC in order to compile ZODB, Zope and lxml.
- libxml2 and libxslt, including development headers.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 or 2.7 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is **admin/admin**.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in `src/` by default, so you can use `mr.developer` to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with `mr.developer` and the checkout command, or you can go directly to editing `checkouts.cfg`. We recommend the latter but will describe both. In the end, `checkouts.cfg` must be configured either way so you might as well start there.

At the base of your buildout, open `checkouts.cfg` and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's **`bin/develop`** command to get the latest source. For example, if the issue is in `plone.app.caching` and `plone.caching`:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, `mr.developer` will download the source from github (or otherwise) and put the package in the `src/` directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with `mr.developer`, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are **ALMOST** ready to commit the changes. A couple housekeeping things before moving on.

First, please edit `CHANGES.rst` (or `CHANGES.txt`, or `HISTORY.txt`) in each package you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers to be able to see what they will get if they upgrade. New changelog entries should be added at the very top of `CHANGES.txt`.

Most importantly, if you didn't do it earlier, edit `checkouts.cfg` file in the buildout directory and add your changes package to the `auto-checkout` list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying `checkouts.cfg` file also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the `checkouts.cfg` file.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update `CHANGES.rst` in each packages you touched?
- Did you add your changed packages to `checkouts.cfg`?

If you answered *YES* to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a `pull request` (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to `roll back` any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).:

```
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Committing to Products.CMFPlone If you are working a bug fix on `Products.CMFPlone`, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then `cherry-pick` the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the `cherry-pick`.

If your fix involved multiple commits, `cherry-picking` them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement form, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be `master`).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the `plone.app.caching` example. After checking it out with `mr.developer`, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to `push` your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says **"Pull Request"**. This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the `pull request` link and ask for a review.

Note: you still need to update `checkouts.cfg` file in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the `CHANGES.txt` of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.

- Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
- Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note: This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't **"require"** PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like `except: pass` will likely be reverted instantly
- Avoid `tal:on-error`, since this swallows exceptions
- Don't use `hasattr()` - this swallows exceptions, use `getattr(foo, 'bar', None)` instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask `ConflictErrors`, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using `context.plone_utils`, use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a Plone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce its own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should:

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at the following link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list and sending a quick email. Then, follow the simple instructions for reviewing a PLIP. Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews the PLIP and gives feedback
6. Developer addresses concerns in feedback and re-submits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standby for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowledgeable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your PLIP is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, *submit a PLIP*. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the community uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criteria by which the framework team will review your bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

See the Reviewing PLIPs page for more information.

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.

- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested

Creating a New PLIP Branch Create a buildout configuration file for your PLIP in the `plips` folder. Give it a descriptive name, starting with the PLIP number; `plip-1234-widget-frobbing.cfg` for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file `plips/plip-1234-widget-frobbing.cfg`:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
    plone.somepackage
    plone.app.someotherpackage

[sources]
plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbing
plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-wi

[instance]
eggs +=
    plone.somepackage
    plone.app.someotherpackage
zcml +=
    plone.somepackage
    plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called `plip_<number>_notes.txt`. This should include (but is not limited to):

- URLs pointing to all documentation created / updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guidelines listed at Reviewing PLIPs.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIPs coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy `mr.developer` is never unhappy, except when it is. Although this technically isn't a buildout issue, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say `allow-hosts =` and `allow-hosts +=`. In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (:file:`.mr.developer.cfg` not found).`

When running any **`./bin/develop`** command.

To fix, simply do:

```
ln -s plips/.mr.developer.cfg
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix `mr.developer` is complaining because a file has been changed/added, but not committed.

Use **`bin/develop update --force`**. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete `mkzopeinstance.py` from `bin/` and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run **`bin/buildout ...`** after **`bin/develop ...`**. Try removing `parts/*`, `bin/*`, `.installed.cfg`, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL `pil.cfg` is include within this buildout to aid in PIL installation. Run **`bin/buildout -c pil.cfg`** to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues **`bin/develop status`** is showing that the `Products.CMFActionIcons` egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit `~/.subversion/config` and add eggtest*.egg to the list of global-ignores

How to Update these Docs These documents are currently stored with the coredev buildout in github in `/docs`. To update them, please checkout the coredev buildout and update there. Make the changes on the latest version branch (as of this writing `4.4`):

```
> git clone git@github.com:plone/buildout.coredev.git
> cd buildout.coredev
> git checkout 4.4
```

To test your changes locally, re-run buildout and then:

```
> bin/sphinx-build docs docs/build
```

Sphinx will poop out a directory that you can put in your browser to validate. For example: `file:///home/user/buildout.coredev/docs/build/index.html`

Please make sure to validate all warnings and errors before committing to make sure the documents remain valid. Once everything is ready to go, commit and push changes.

Cherry pick commits on the latest branch to the currently released branch (as of this writing `4.3`) if these changes apply to that version (you can get the SHA hash from **`git log`**):

```
> git checkout 4.3
> git cherry-pick b6ff4309
```

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the **`git cherry-pick`**.

Others These are some documents using as reference for this documentation.

Contributor's Agreement for Plone Explained Prospective contributors to the Plone core code base are required to sign a contributor's agreement, which assigns copyright in the code to the Plone Foundation, the non-profit organization which stewards the Plone code base. This document explains the purposes of this, along with questions and answers about what this means.

The Plone Contributor Agreement can be found at: http://plone.org/foundation/contributors-agreement/agreement.pdf

About the Plone Contributor Agreement The Foundation feels that it benefits the community for a single organization to hold the rights to Plone. Prior to the Foundation, the intellectual property of Plone was jointly held by individual developers and by Alan Runyan and Alexander Limi. The community members who formed the Foundation felt that having the Foundation hold these rights provides several benefits:

1. **Minimizing confusion / maximizing business compatibility** – Organizations considering adopting Plone have a simple answer for "Who owns this?", rather than a more complicated answer that might scare away the legally-cautious.
2. **Trademark protection** – By having the Foundation hold the trademarks and rights to the Plone branding assets, it can effectively protect these from unfair use.

3. **Guarantee of future Open Source versions** – The Foundation's contributor agreement ensures that there will **always** be an OSI- approved version of Plone.

Questions & Answers What does the contributor's agreement cover?

> This agreement is for the Plone core codebase only. The Plone core codebase is that code which lives in the Plone core version repositories, currently located at http://dev.plone.org/plone and https://github.com/plone. Contributions to the "Collective," currently located at http://dev.plone.org/collective and https://github.com/collective are not assigned to the Plone Foundation, and are made available under whatever license the project developers wish to use, although add-on products that import from GPLed Plone code are of course subject to the terms of the GPL, which requires derived works to be GPL licensed.

What rights will I continue to have for my contributions?

> Contributors are asked to transfer their intellectual property rights to the Foundation. In return, they will be given back irrevocable rights to use and distribute their contributions. They can even give their contributions to other Open Source projects (as long as those projects are compatible with the license Plone itself is issued under) or use them in non-Open Source commercial applications (if that is compatible with the license Plone is under).

Do I have to sign the contributor's agreement to make checkins to the Plone core codebase? Yes.

Do I have to sign the contributor's agreement to submit a patch to the Plone core codebase?

> We enthusiastically welcome patches, but we can't merge them until you sign and return a contributor's agreement. (Unless, in the judgement of the Plone Release Manager, the patch is so tiny as not to constitute a "creative work." See the Policy for Contributor Agreements and Patches for more detail on this policy.)

Can I grant the Plone foundation a non-exclusive license to my contributions rather than an exclusive license, so that I can contribute the same code to other projects under different terms or use the contribution for other commercial endeavors?

> Not under the current version of the contributors agreement.

Does the Foundation control use of the Plone trademark?

> Yes. In order to keep the trademark, the Foundation (or any trademark owner) must demonstrate that they have acted to protect it.

Will Plone always be available under an OSI-approved/Open Source license? Couldn't the Board change its mind about this?

> Plone will always be available under an OSI-approved license; this is written into the language of the contributor agreement each developer and the foundation sign.

Will Plone ever be available under a non-GPL license?

> The current Plone approach states that companies can negotiate a non-GPL license. Thus, the Foundation might pursue a dual-licensing (GPL and non-GPL) scheme - but, at this time, the Board has not yet created any policies on this. This is an important question for the community, of course, and the Foundation intends to have this conversation in a transparent way.

Why would anyone want a non-GPL Plone?

> Two possible reasons: Some companies are reluctant to do in-house modifications of framework-like systems (such as Plone) that are under the GPL, fearing that a clause in the GPL might force them to disclose their internal work - thus wanting to license it under (for example) a BSD-style license. Second, companies may wish to offer a commercial version of Plone, under a conventional shrink-wrap license, without the obligation to reveal source code or share changes.

How much would a non-GPL version of Plone cost?

> Would a small company be able to afford one? – Neither the Foundation nor the Board have made any decisions about a non-GPL version, let alone about pricing. However, one of the Foundation's stated goals is to maintain a level playing field for Plone while trying to benefit all of the Plone commons. If a non-GPL version was available, and a large company bought it, added features to it, and sold it, wouldn't they be using our work without an obligation to give back? It's helpful to remember the core value open source provides: distributed development, maintenance, security checking, and support. Companies that build large features for Plone are **already** having to make decisions of whether to release their products under an open source license or not (since they could always release them as a Product, not as a modification to the Plone core). Despite this, though, many large and excellent contributions - such as Archetypes - have been made, and the Foundation hopes that companies will continue to do so. In any event, a company that purchases a non-GPL license (should such ever become available) is contributing financial resources to our community, which can be used to further develop, market, and protect the GPL version of Plone.

Mr. Developer This buildout uses mr.developer to manage package development. See http://pypi.python.org/pypi/mr.developer for more information or run **`bin/develop help`** for a list of available commands.

The most common workflow to get all the latest updates is:

```
> git pull
> bin/develop rb
```

This will get you the latest coredev configuration, checkout and update all packages via Subversion in src and run buildout to configure the whole thing.

From time to time you can check if some old cruft has accumulated:

```
> bin/develop st
```

If this prints any lines with a question mark in front, you can cleanup by:

```
> bin/develop purge
```

This will remove packages from `src/` which are no longer needed, as they have been replaced by proper egg releases of these packages.

Reviewing PLIPs

Expectations A good PLIP review takes about 4 hours so please plan accordingly. When you are done, if you have access to core please commit the review to the plips folder and reference the PLIP in your commit message. If you do not have access, please attach your review to the PLIP ticket itself.

Setting up the environment Follow the instructions on setting up a development environment for "Getting the Code". You will need to checkout the branch to which the PLIP is assigned. Instead of running the buildout with the default buildout file, you will run the config specific to that plip:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Functionality Review There are several things that could be addressed in a PLIP review depending on the nature of the PLIP itself. This is by no means an exhaustive list, but a place to start. Things to think about when reviewing:

General

- Does the PLIP actually do what the implementors proposed? Are there incomplete variations?
- Were there any errors running buildout? Did the migration(s) work?
- Do error and status messages make sense? Are they properly internationalized?
- Are there any performance considerations? Has the implementor addressed them if so?

Bugs

- Are there any bugs? Nothing is too big nor small.
- Do fields handle whacky data? How about strings in date fields or nulls in required?
- Is validation up to snuff and sensical? Is it too restrictive or not restrictive enough?

Usability Issues

- Is the implementation usable?
- How will novice end users respond to the change?
- Does this PLIP need a usability review? If you think this PLIP needs a usability review, please change the state to "please review" and add a note in the comments.
- Is the PLIP consistent with the rest of Plone? For example, if there is control panel configuration, does the new form fit in with the rest of the panels?
- Does everything flow nicely for novice and advanced users? Is there any workflow that feels odd?
- Are there any new permissions and do they work properly? Does their role assignment make sense?

Documentation Issues

- Is the corresponding documentation for the end user, be it developer or plone user, sufficient?
- Is the change itself properly documented?

Please report bugs/issues in Trac as you would for any Plone bug. Reference the PLIP in the bug, assign to its implementor, and add a tag for the PLIP in the form of plip-xxx. This way the implementor can find help if he needs it. Please also prioritize the ticket. The PLIP will not be merged until all blockers and critical bugs are fixed.

Code Review

Python

- Is this code maintainable?
- Is the code properly documented?
- Does the code adhere to PEP8 standards (more or less)?
- Are they importing deprecated modules?

Javascript

- Does the javascript meet our set of javascript standards? See http://developer.plone.org/templates_css_and_javascripts/javascript.ht
- Does the Javascript work in all currently supported browsers? Is it performant?

ME/TAL

- Does the PLIP use views appropriately and avoiding too much logic?
- Is there any code in a loop that could potentially be a performance issue?
- Are there any deprecated or old style ME/TAL lines of code such as using DateTime?
- Is the rendered html standards compliant? Are ids and classes used appropriately?

Example PLIP Reviews

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Translations available There are some initiatives to maintenance available translations in other languages for this documentation like:

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.

3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.

2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.

- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes

- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage

people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems

with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout ...' after 'bin/develop ...'. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout ...' after 'bin/develop ...'. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout ...' after 'bin/develop ...'. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

README.rst The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

CHANGES.txt The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise

- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that

needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development

- Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/*, bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).

- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference

  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.

- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.

7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

Process for Plone core's development

Introdução Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO! Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL

e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...
- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture

- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string
- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
    from Products.CMFCore.utils import getToolByName
    plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-

[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 55
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force`. Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt

1.4.2 Process for Plone core's development

Introdução

Esta documentação descreve o processo para desenvolvimento usando o Plone e é um recurso técnico para configurar seu core developer buildout, corrigir erros e escrever PLIP's (PLone Improvement Proposal - Propostas de Melhoria do Plone).

ATENÇÃO!

Legalmente, você NÃO pode contribuir com código sem que tenha assinado o acordo de contribuição; sem ele, não podemos aceitar seus pull requests. Enviar o acordo é fácil e será mais fácil em breve. Por isto, não coloque os revisores do código em risco, de forma alguma. Se você quiser acesso rápido e estiver familiarizado com a comunidade, acesse irc e peça para algum administrador do repositório lhe dar acesso, enviando uma cópia escaneada do acordo. Eles permitirão que você comece a contribuir o mais rápido possível.

Sumário

Acordo de Contribuição Contribuidores em potencial para a base de código do núcleo do Plone devem assinar o acordo de contribuição, que atribui os direitos autorais do código para a Plone Foundation, a organização sem fins lucrativos que administra a base de código do Plone. Este documento explica os objetivos do acordo, assim como perguntas e respostas sobre o que isto significa.

O acordo de contribuição do Plone pode ser encontrado em: http://plone.org/foundation/contributors-agreement/agreement.pdf

Sobre o acordo de contribuição A Fundação considera que é vantajoso para a comunidade uma única organização manter os direitos sobre o Plone. Antes da Fundação, a propriedade intelectual do Plone era mantida conjuntamente por desenvolvedores individuais e por Alan Runyan e Limi Alexander. Os membros da comunidade que formaram a Fundação entenderam que a Fundação manter esses direitos ofereceria vários benefícios:

1. **Reduzir confusão/aumentar a compatibilidade de negócio** – Organizações que estejam considerando adotar o Plone recebem uma resposta simples para "A quem pertence o Plone?", ao invés de uma resposta complexa que poderia intimidar os mais legalistas.
2. **Proteção da marca** – Como a Fundação mantém direitos sobre as marcas e elementos do Plone, ela pode efetivamente protegê-los contra uso indevido.
3. **Garantia de futuras versões Open Source** – O acordo de contribuição da Fundação garante que **sempre** haverá uma versão do Plone aprovada pela OSI (Open Source Iniciative - Iniciativa Open Source).

Perguntas e Respostas

O que o acordo de contribuição cobre? Este acordo se aplica somente ao núcleo do código do Plone, que é o código mantido nos repositórios da versão núcleo do Plone e está localizado atualmente em http://dev.plone.org/plone e https://github.com/plone. Contribuições para o "Collective", atualmente localizado em http://dev.plone.org/collective e https://github.com/collective, não são atribuídas à Plone Foundation, e são disponibilizadas sob qualquer licenciamento que o desenvolvedor do projeto deseje, embora produtos que importem código do Plone sob a GPL estejam sujeitos aos termos da GPL, que requer que trabalhos derivados também sejam licenciados sob a GPL.

Que direitos manterei sobre minhas contribuições? Contribuidores devem transferir seus direitos sobre a propriedade intelectual para a Fundação. Em troca, eles receberão direitos irrevogáveis de uso e distribuição sobre suas contribuições, podendo inclusive ceder suas contribuições para outros projetos Open Source, desde que os projetos sejam compatíveis com a licença utilizada pelo Plone, ou utilizá-los em aplicações comerciais não-Open Source, se estas forem compatíveis com a licença do Plone.

Eu preciso assinar o acordo de contribuição para poder fazer checkins na base de código do Plone? Sim.

Eu preciso assinar o acordo de contribuição para poder enviar um patch para a base de código do Plone? Patches são entusiasticamente bem-vindos, mas não poderemos mesclá-los até que você assine e envie o acordo de contribuição. (Exceto caso o Gerenciador de Release do Plone considere que o patch é tão pequeno que não constitua um "trabalho criativo". Consulte a Política para acordos de contribuição e patches (em inglês) para mais detalhes.)

Eu posso conceder à Plone Foundation uma licença não-exclusiva sobre minhas contribuições, ao invés de uma licença exclusiva
Não, de acordo com a versão atual do acordo de contribuição.

A Fundação controla o uso da marca Plone? Sim. Para manter a marca, a Fundação (ou qualquer outro proprietário de marca) deve demonstrar que atuou de forma a protegê-la.

O Plone sempre estará disponível sob uma licença aprovada pela OSI/Open Source? O Conselho não pode mudar de ideia sobre
O Plone sempre estará disponível sob uma licença aprovada pela OSI; isto está escrito no acordo de contribuição que cada desenvolvedor e a Fundação assinam.

O Plone será disponibilizado sob uma licença não-GPL? A abordagem atual do Plone define que as empresas podem negociar uma licença não-GPL. Assim, a Fundação pode prover um esquema de licenciamento duplo (GPL e não-GPL) - mas o Conselho ainda não criou quaisquer políticas sobre este assunto. Esta é uma questão importante para a comunidade, é claro, e a Fundação pretende tratar este assunto de forma transparente.

Por que alguém iria querer um Plone não-GPL? Duas razões possíveis: Algumas empresas relutam em fazer mudanças em sistemas semelhantes a frameworks (como o Plone) licenciadas sob a GPL, temendo serem forçadas a abrir seu trabalho interno em função de alguma cláusula da GPL - podendo assim licenciá-las sob uma licença no estilo BSD, por exemplo. Além disso, empresas podem desejar oferecer uma versão comercial do Plone, sob uma licença hermética, sem a obrigação de revelar seu código fonte ou compartilhar alterações.

Cultura do desenvolvedor Plone Se você deseja contribuir com o desenvolvimento do Plone, siga algumas instruções:

1. Cadastre-se na lista plone-developers e veja como as pessoas trabalham e como costumam abordar as discussões no grupo.
2. Peça ajuda sobre configuração do ambiente no IRC. A maioria dos desenvolvedores estão lá.

Baixe um cliente de IRC e acesse o canal #plone-framework ou #plone, ambos no freenode. As pessoas no #plone-framework usam Plone há muito tempo e ficam felizes em poder ajudar. Mais informações podem ser obtidas no endereço http://plone.org/support/chat.

Se você está em um fuso horário em que o canal não esteja muito ativo, envie um e-mail para a lista plone-developers ou pegue uma cerveja e espere o pessoal acordar.

Quando tiver dúvidas, pode perguntar. O código do Plone é muito complicado e todos estão empenhados em fazer a coisa certa acontecer. Apesar de uma ou outra grosseria, a maioria dos desenvolvedores Plone irá deixar o que está fazendo para lhe ajudar a achar o caminho certo.

Como enviar correções para o Core do Plone Este documento assume que você deseja corrigir um bug e irá detalhar o processo completo. Para mais informações sobre PLIPS, go here.

Política de Versões Suportadas Se você estiver fazendo uma triagem ou corrigindo bugs, tenha em mente que o Plone possui uma Política de versões suportadas.

Dependências

- Git
- Subversion
- Python 2.6 or 2.7 incluindo as bibliotecas de desenvolvimento.
- If you are on Mac OSX, you will need to install XCode. You can do this through the app store or several other soul-selling methods. You will likely want to install your own python 2.6 as well since they strip out all the header files which makes compiling some extensions weird. You can ignore this advice to start, but have faith, you'll come back to it later. They always do...

- Python Imaging Library (PIL). Certifique-se de instalar no ambiente Python adequado.
- VirtualEnv No ambiente Python adequado.
- GCC A fim de compilar ZODB, Zope e lxml.
- libxml2 and libxslt, Incluíndo as bibliotecas de desenvolvimento.

Setting up Your Development Environment The first step in fixing a bug is getting this buildout running. We recommend fixing the bug on the latest branch and then backporting as necessary. Github by default always points to the currently active branch. More information on switching release branches is below.

To set up a plone 4.2 development environment:

```
> cd ~/buildouts # or wherever you want to put things
> git clone -b 4.2  https://github.com/plone/buildout.coredev ./plone42devel
> virtualenv --no-site-packages plone42devpy
> cd plone42devel
> ../plone42devpy/bin/python bootstrap.py # (where "python" is your python 2.6 binary).
> bin/buildout -v
```

If you run into issues in this process, please see the doc Troubleshooting.

This will run for a long time if it is your first pull (~20 mins). Once that is done pulling down eggs, You can start your new instance with:

```
> ./bin/instance fg
```

The default username/password for a dev instance is admin/admin.

Switching Branches If your bug is specific to one branch or you think it should be backported, you can easily switch branches. The first time you get a branch, you must do:

```
> git checkout -t origin/4.1
```

This should set up a local 4.1 branch tracking the one on github. From then on you can just do:

```
> git checkout 4.1
```

To see what branch you are currently on, just do:

```
> git branch
```

The line with a * by it will indicate which branch you are currently working on.

Important: Make sure to rerun buildout if you were in a different branch earlier to get the correct versions of packages, otherwise you will get some weird behavior!

For more information on buildout, please see the collective developer manual documentation on buildout.

Checking out Packages for Fixing Most packages are not in src/ by default, so you can user mr.developer to get the latest and make sure you are always up to date. It can be a little daunting at first to find out which packages are causing the bug in question, but just ask on irc if you need some help. Once you [think you] know which package(s) you want, we need to pull the source.

You can get the source of the package with mr.developer and the checkout command, or you can go directly to editing checkouts.cfg. We recommend the latter but will describe both. In the end, checkouts.cfg must be configured either way so you might as well start there.

At the base of your buildout, open checkouts.cfg and add your package if it's not already there:

```
auto-checkout =
        # my modified packages
        plone.app.caching
        plone.caching
        # others
        ...
```

Then rerun buildout to get the source packages:

```
> ./bin/buildout
```

Altternatively, we can manage checkouts from the command line, by using mr.developer's `bin/develop` command to get the latest source. For example, if the issue is in plone.app.caching and plone.caching:

```
> ./bin/develop co plone.app.caching
> ./bin/develop co plone.caching
> ./bin/buildout
```

Don't forget to rerun buildout! In both methods, mr.developer will download the source from github (or otherwise) and put the package in the src directory. You can repeat this process with as many or as few packages as you need. For some more tips on working with mr.developer, please read more here.

Testing Locally In an ideal world, you would write a test case for your issue before actually trying to fix it. In reality this rarely happens. No matter how you approach it, you should ALWAYS run test cases for both the module and plone.org before commiting any changes.

If you don't start with a test case, save yourself potential problems and validate the bug before getting too deep into the issue!

To run a test for the specific module you are modifying:

```
> ./bin/test -m plone.app.caching
```

These should all run without error. Please don't check in anything that doesn't! If you haven't written it already, this is a good time to write a test case for the bug you are fixing and make sure everything is running as it should.

After the module level tests run with your change, please make sure other modules aren't affected by the change by running the full suite:

```
> ./bin/alltests
```

Note: Tests take a long time to run. Once you become a master of bugfixes, you may just let jenkins do this part for you. More on that below.

Updating CHANGES.rst and checkouts.cfg Once all the tests are running locally on your machine, you are ALMOST ready to commit the changes. A couple housekeeping things before moving on.

First, please edit CHANGES.rst (or CHANGES.txt) in each pakage you have modified and add a summary of the change. This change note will be collated for the next Plone release and is important for integrators and developers.

Most importantly, if you didn't do it earlier, edit checkouts.cfg in the buildout directory and add your changes package to the auto-checkout list. This lets the release manager know that the package has been updated so that when the next release of Plone is cut a new egg will be released and Plone will need to pin to the next version of that package. READ: this is how your fix becomes an egg!

Note that there is a section seperator called "# Test Fixes Only". Make sure your egg is above that line or your egg probably won't get made very quickly. This just tells the release manager that any eggs below this line have tests that are updated, but no code changes.

Modifying checkouts.cfg also triggers the buildbot, jenkins, to pull in the egg and run all the tests against the changes you just made. Not that you would ever skip running all tests of course... More on that below.

If your bug is in more than one release (e.g. 4.1 and 4.2), please checkout both branches and add to the checkouts.cfg.

Committing and Pull Requests Phew! We are in the home stretch. How about a last minute checklist:

- Did you fix the original bug?
- Is your code consistent with our Style Guide?
- Did you remove any extra code and lingering pdbs?
- Did you write a test case for that bug?
- Are all test cases for the modules(s) and for Plone passing?
- Did you update CHANGES.rst in each packages you touched?
- Did you add your changed packages to checkouts.cfg?

If you answered YES to all of these questions, you are ready to push your changes! A couple quick reminders:

- Only commit directly to the development branch if you're confident your code won't break anything badly and the changes are small and fairly trivial. Otherwise, please create a pull request (more on that below).
- Please try to make one change per commit. If you are fixing three bugs, make three commits. That way, it is easier to see what was done when, and easier to roll back any changes if necessary. If you want to make large changes cleaning up whitespace or renaming variables, it is especially important to do so in a separate commit for this reason.
- We have a few angels that follow the changes and each commit to see what happens to their favourite CMS! If you commit something REALLY sketchy, they will politely contact you, most likely after immediately reverting changes. There is no official people assigned to this so if you are especially nervous, jump into #plone and ask for a quick eyeball on your changes.

Committing to Products.CMFPlone If you are working a bug fix on Products.CMFPlone, there are a couple other things to take notice of. First and foremost, you'll see that there are several branches. At the time of writing this document, there are branches for 4.1, 4.2, and master, which is the implied 4.3.

Still with me? So you have a bug fix for 4.x. If the fix is only for one version, make sure to get that branch and party on. However, chances are the bug is in multiple branches.

Let's say the bug starts in 4.1. Pull the 4.1 branch and fix and commit there with tests.

If your fix only involved a single commit, you can use git's `cherry-pick` command to apply the same commit to a different branch.

First check out the branch:

```
> git checkout 4.2
```

And then cherry-pick the commit (you can get the SHA hash from git log).

> git cherry-pick b6ff4309

There may be conflicts; if so, resolve them and then follow the directions git gives you to complete the cherry-pick.

If your fix involved multiple commits, cherry-picking them one by one can get tedious. In this case things are easiest if you did your fix in a separate feature branch.

In that scenario, you first merge the feature branch to the 4.1 branch:

```
> git checkout 4.1
> git merge my-awesome-feature
```

Then you return to the feature branch and make a branch for *rebasing* it onto the 4.2 branch:

```
> git checkout my-awesome-feature
> git checkout -b my-awesome-feature-4.2
> git rebase ef978a --onto 4.2
```

(ef978a happens to be the last commit in the feature branch's history before it was branched off of 4.1. You can look at git log to find this.)

At this point, the feature branch's history has been updated, but it hasn't actually been merged to 4.2 yet. This lets you deal with resolving conflicts before you actually merge it to the 4.2 release branch. Let's do that now:

```
> git checkout 4.2
> git merge my-awesome-feature-4.2
```

Branches and Forks and Direct Commits - Oh My! Plone used to be in an svn repository, so everyone is familiar and accustomed to committing directly to the branches. After the migration to github, the community decided to maintain this spirit. If you have signed the contributor agreement, you can commit directly to the branch (for plone this would be the version branch, for most other packages this would be master).

HOWEVER, there are a few situations where a branch is appropriate. If you:

- are just getting started,
- are not sure about your changes
- want feedback/code review
- are implementing a non-trivial change

then you likely want to create a branch of whatever packages you are using and then use the pull request feature of github to get review. Everything about this process would be the same except you need to work on a branch. Take the plone.app.caching example. After checking it out with mr.developer, create your own branch with:

```
> cd src/plone.app.caching
> git checkout -b my_descriptive_branch_name
```

Note: Branching or forking is your choice. I prefer branching, and I'm writing the docs so this uses the branch method. If you branch, it helps us because we *know* that you have committer rights. Either way it's your call.

Proceed as normal. When you are ready to push your fix, push to a remote branch with:

```
> git push origin my_descriptive_branch_name
```

This will make a remote branch in github. Navigate to this branch in the github UI and on the top right there will be a button that says "Pull Request". This will turn your request into a pull request on the main branch. There are people who look once a week or more for pending pull requests and will confirm whether or not its a good fix and give you feedback where necessary. The reviewers are informal and very nice so don't worry - they are there to help! If you want immediate feedback, jump into IRC with the pull request link and ask for a review.

Note: you still need to update checkouts.cfg in the correct branches of buildout.coredev!

Jenkins You STILL aren't done! Please check jenkins to make sure your changes haven't borked things. It runs every half an hour and takes a while to run so checking back in an hour is a safe bet. Have a beer and head over to the Jenkins control panel.

Finalizing Tickets If you are working from a ticket, please don't forget to go back to the ticket and add a link to the changeset. We don't have integration with github yet so it's a nice way to track changes. It also lets the reporter know that you care. If the bug is really bad, consider pinging the release manager and asking him to make a release pronto.

FAQ

- ***How do I know when my package got made?*** You can follow the project on github and watch its timeline. You can also check the CHANGES.txt of every plone release for a comprehensive list of all changes and validate that yours is present.

Writing documentation

Documentation of Plone As a community, Plone maintains several types of documentation:

- *Curated* documents. This is a limited set of documentation that is intended to be carefully managed and regularly updated.
  - User Manual
  - Installing Plone
  - Theme Reference
  - Developer Manual

  Improvements to the curated documents can be discussed on the plone-docs mailing list.
- *Community-edited* documents. These are open for contributions by anyone. This leads to a wealth of information that is of more widely ranging quality.
  - Knowledgebase on plone.org. Anyone with a plone.org account is free to edit.
  - Collective Plone developer documentation. Anyone may contribute.

Documenting a package

The basics At the very least, your package should include the following forms of documentation:

`README.rst` The readme is the first entry point for most people to your package. It will be included on the PyPI page for your egg, and on the front page of its github repository. It should be formatted using reStructuredText (reST) in order to get formatted properly by those systems.

`README.rst` should include:

- A brief description of the package's purpose
- Installation information (How do I get it working?)
- Compatibility information (what versions of Plone does it work with?)
- Links to other sources of documentation
- Links to issue trackers, mailing lists, and other ways to get help.

The manual (a.k.a. narrative documentation)

The manual goes into further depth for people who want to know all about how to use the package.

It includes topics like:

- What the features are
- How to use them (in English—not doctests!)
- Information about architecture
- Common gotchas

The manual should consider various audiences who may need different types of information:

- End users who use Plone for content editing but don't manage the site.
- Site administrators who install and configure the package.
- Integrators who need to extend the functionality of the package in code.
- Sysadmins who need to maintain the server running the software.

Simple packages with limited functionality can get by with a single page of narrative documentation. In this case it's simplest to include it in an extended `README.rst`. Some excellent examples of a single-page readme are http://pypi.python.org/pypi/plone.outputfilters and https://github.com/plone/plone.app.caching

If your project is moderately complex, you may want to set up your documentation with multiple pages. The best way to do this is to add Sphinx to your project and host your docs on readthedocs.org so that it rebuilds the documentation whenever you push to github. If you do this, your `README.rst` must link off site to the documentation.

Reference (a.k.a. API documentation)

An API reference provides information about the package's public API (that is, the code that the package exposes for use from external code.) It is meant for random access to remind the reader of how a particular class or method works, rather than for reading in its entirety.

If the codebase is written with docstrings, API documentation can be automatically generated using Sphinx.

`CHANGES.txt` The changelog is a record of all the changes made to the package and who made them, with the most recent changes at the top. This is maintained separately from the git commit history to give a chance for more user-friendly messages and to and record when releases were made.

A changelog looks something like:

```
Changelog
=========

1.0 (2012-03-25)
----------------

* Documented changelogs.
  [davisagli]
```

See https://raw.github.com/plone/plone.app.caching/master/CHANGES.rst for a full example.

If a change was related to a bug in the issue tracker, the changelog entry should include a link to that issue.

Licenses Information about the open source license used for the package should be placed within the `docs` directory.

For Plone core packages, this includes `LICENSE.txt` and `LICENSE.GPL`.

Using Sphinx

reST References:

- Plone Oriented Shpinx Documentation
- Sphinx reST Primer

To add Sphinx to your package...

Style Guide Python, like any programming language, can be written in a number of styles. We're the first to admit that Zope and Plone are not the finest examples of stylistic integrity, but that doesn't stop us from trying!

If you are not familiar with PEP 8 - the python style guide, please take a moment to read and get up to date. We don't require it but we as a community really, really appreciate it.

Naming Conventions Above all else, be consistent with any code your are modifying! Historically the code is all camel case, but many new libraries are in the PEP8 convention. The mailing list is exploding with debate over what is better so we'll leave the excersize of deciding what to do with the user.

File Conventions In Zope 2, file names used to be MixedCase. In Python, and thus in Plone going forward, we prefer all-lowercase filenames. This has the advantage that you can instantly see if you refer to a module / file or a class:

```
from zope.pagetemplate.pagetemplate import PageTemplate
```

compare that to:

```
from Products.PageTemplates.PageTemplate import PageTemplatePageTemplate
```

Filenames should be short and descriptive. Think about how an import would read:

```
from Products.CMFPlone.utils import safe_hasattr
```

compare that to:

```
from Products.CMFPlone.PloneUtilities import safe_hasattr
```

The former is obviously much easier to read, less redundant and generally more aesthetically pleasing.

Note This example is just about as terrible as they come. We need a better one.

Concrete Rules

- Do not use tabs in Python code! Use spaces as indenting, 4 spaces for each level. We don't "require" PEP8, but most people use it and it's good for you.
- Indent properly, even in HTML.
- Never use a bare except. Anything like 'except: pass' will likely be reverted instantly
- Avoid tal:on-error, since this swallows exceptions
- Don't use hasattr() - this swallows exceptions, use getattr(foo, 'bar', None) instead. The problem with swallowed exceptions is not just poor error reporting. This can also mask ConflictErrors, which indicate that something has gone wrong at the ZODB level!
- Never, ever put any HTML in Python code and return it as a string

- Do not acquire anything unless absolutely necessary, especially tools. For example, instead of using 'context.plone_utils', use:

```
from Products.CMFCore.utils import getToolByName
plone_utils = getToolByName(context, 'plone_utils')
```

- Do not put too much logic in ZPT (use Views instead!)
- Remember to add i18n tags in ZPTs and Python code

Implementing PLIPS

All about PLIPS

What is a PLIP? A PLIP is a PLone Improvement Proposal. It is a change to a Plone package that would affect everyone. PLIPs go through a different process than bug fixes because of their broad reaching effect. The Plone 4.x Framework Team reviews all PLIPs to be sure that it's in the best interest of the broader community to be implemented and that it is of high quality.

Is it a PLIP or a bugfix? In general, anything that changes the API of Plone in the backend or UI on the front end should be filed as a PLIP. When in doubt, submit it as a PLIP. The framework team is eager to reduce it's own workload and will re-classify it for you.

Who can submit PLIPs? Anyone who has signed a Plone core contributor agreement can work on a PLIP. Don't let the wording freak you out: signing the agreement is easy and you will get access almost immediately. You do not have to be the most amazing coder in the entire world to submit a PLIP. The Framework Team is happy to help you at any point in the process. Submitting a PLIP can be a great learning process and we encourage people of all backgrounds to submit. When the PLIP is accepted, a Framework Team member will "champion" your PLIP and be dedicated to seeing it completed. PLIPs are not just for code monkeys. If you have ideas on new interactions or UI your ideas are more than welcome. We will even help you pair up with implementors if needed.

What is a PLIP champion? When you submit your PLIP and it is approved, 1 Framework Team member who is especially excited about seeing the PLIP completed will be assigned to your PLIP as a champion. They are there to push you through completion as well as answer any questions and provide guidance.

A champion should

- Answer any questions the PLIP implementor has, technically and otherwise
- Encourage the PLIP author by constantly giving feedback and encouragement
- Keep the implementor aware of timelines and push to get things done on time
- Assist with finding additional help when needed to complete the implementation in a timely matter

Keep in mind that champions are in passive mode by default. If you need help or guidance, please reach out to them as soon as possible to activate help mode.

I'm still nervous. Can I get involved other ways at first? If you want to feel the process and how it works, help us review PLIPs as the implementations finish up. Simply ask on of the Framework Team members what PLIPs are available for review or check the status of PLIPs at XXX:Link. Make sure to let us know you intend to review the PLIP by joining the Framework Team mailing list (XXX:link) and sending a quick email. Then, follow the simple instructions for reviewing a PLIP (XXX:link). Thank you in advance!

When can I submit a PLIP? Today, tomorrow, any time! After the PLIP is accepted, the Framework Team will try to judge complexity and time to completion and assign it to a milestone. You can begin working immediately, and we encourage submitting fast and furious.

When is the PLIP due? Summary: As soon as you get it done. &tldr; Technically, we want to see it completed for the release to which it's assigned. We know that things get busy and new problems make PLIPs more complicated and we will push it to the next release. In general, we don't want to track a PLIP for more than a year. If your PLIP is accepted and we haven't seen activity in over a year, we will probably ask you to restart the whole process.

You don't like my PLIP :(What now? Just because a PLIP isn't accepted in core doesn't mean it's a bad idea. It is often the case that there are competing implementations and we want to see it vetted as an add on before "blessing" a preferred implementation.

Process Overview

1. Submit a PLIP (at any time)
2. PLIP is approved for inclusion into core for a given release
3. Developer implements PLIP (code, tests, documentation)
4. PLIP is submitted for review by developer
5. Framework Team reviews plip and givea feedback
6. Developer addresses concerns in feedback and resubmits if necessary. This may go back and forth a few times until both the FWT and developer are happy with the result.
7. PLIP is approved for merge. In rare circumstances, a PLIP will be rejected. This is usually the result of the developer not responding to feedback or dropping out of the process. Hang in there!
8. After all other PLIPS are merged, a release is cut. Standy for bugs!

How to Submit a PLIP Whether you want to update the default theme or rip out a piece of architecture, everyone should go through the PLIP process. If you need help at any point in this process, please contact a member of the framework team personally or ask for help on the FWT mailing list XXX: link here.

A PLIP is just a ticket with a special template. To get started, open a new ticket and select "PLIP" as the ticket type. A new ticket template will reload and you should plan to fill in all of the fields.

When writing a PLIP, be as specific and to-the-point as you can. Remember your audience - to get support for your proposal, people will have to be able to read it! A good PLIP is sufficiently clear for a knowlegable Plone user to be able to understand the proposed changes, and sufficiently detailed for the release manager and other developers to understand the full impact the proposal would have on the codebase. You don't have to list every line of code that needs to be changed, but you should also give an indication that you have some idea that how the change can be feasibly implemented.

If your change is minor then a ticket in the tracker will be sufficient, added as an enhancement. The key point here is that each change needs documentation so other users can see what it is. This can be in the form of an issue tracker entry, or a PLIP in the case of a bigger change. A bug or minor change does normally not need to go through a review process - a PLIP does.

After your plip is written, solicit feedback on your idea on the plone-developers mailing list. In this vetting process, you want to make sure that the change won't adversely affect other people on accident. Others may be able to point out risks or even offer up better or existing solutions.

When you are happy with the feedback, submit a PLIP. XXX: Link here. Please use the template provided (XXX: put the template here? Can we just have a custom ticket type?). Please note a few things. It is very rare that the "Risks" section will be empty or none. If you find this is the case and your PLIP is anything more than trivial, maybe some more vetting should be done.

The seconder field is REQUIRED. We will send the PLIP back to you if it is not filled in. Currently, this is just someone else who thinks your PLIP is a good idea, a +1. In the near future, we will start asking that the seconder

is either a coding partner, or someone who is willing and able to finish the PLIP should something happen to the implementor.

Everything else should be self explanatory. That or I got lazy when writing these docs. I'm betting on the latter.

Evaluating PLIPs After you submit your PLIP, the Framework Team will meet within a couple weeks and let you know if the PLIP is accepted. If the PLIP is not accepted, please don't be sad! We encourage most PLIPs to go through the add on process at first if at all possible to make sure the majority of the comunity uses it.

All communication with you occurs on the PLIP ticket itself so please keep your eyes and inbox open for changes.

These are the criterias by which the framework team will review your review bundle:

- What is size and status of the work needed to be done? Is it already an add-on and well established?
- Is this idea well baked and expressed clearly?
- Does the work proposed belong in Plone now, in the future?
- Is this PLIP more appropriate as a qualified add-on?
- Is this PLIP too risky?

Implementing Your PLIP You can start the development at any time - but if you are going to modify Plone itself, you might want to wait to see if your idea is approved first to save yourself some work if it isn't.

General Rules

- Any new packages must be in a branch in the plone namespace in github. You don't have to develop there, but it must be there when submitted. We recommend using branches off of the github.com/plone repo and will detail that below.
- Most importantly, the PLIP reviewers must be able run buildout and everything should "just work" (tm).
- **Any new code must:**
  - Be Properly Documented
  - Have clear code
  - User the current idioms of development
  - Be tested (XXX: link here?)

Creating a New PLIP Branch Create a buildout configuration file for your plip in the 'plips' folder. Give it a descriptive name, starting with the plip number; 'plip-1234-widget-frobbing.cfg' for example. This file will define the branches/trunks you're working with in your PLIP. It should look something like this:

In file plips/plip-1234-widget-frobbing.cfg...:

```
[buildout]
extends = plipbase.cfg
auto-checkout +=
 plone.somepackage
 plone.app.someotherpackage

[sources]
 plone.somepackage = git git://github.com/plone/plone.somepackage.git branch=plip-1234-widget-frobbi
 plone.app.someotherpackage = git git://github.com/plone/plone.app.somepackage.git branch=plip-1234-w
```

```
[instance]
eggs +=
   plone.somepackage
   plone.app.someotherpackage
zcml +=
   plone.somepackage
   plone.app.someotherpackage
```

Use the same naming convention when branching existing packages, and you should always be branching packages when working on PLIPs.

Finishing Up Before marking your PLIP as ready for review, please add a file to give a set of instructions to the PLIP reviewer.

This file should be called plip_<number>_notes.txt. This should include (but is not limited to):

- URLs pointing to all documentation created/updated
- Any concerns, issues still remaining
- Any weird buildout things
- XXX: What else?

Once you have finished, please update your PLIP ticket to indicate that it is ready for review. The Framework Team will assign 2-3 people to review your PLIP. They will follow the guildelines listed at Revisando PLIP's.

After the PLIP has been accepted by the framework team and the release manager, you will be asked to merge your work into the main development line. Merging the PLIP in is not the hardest part, but you must think about it when you develop. You'll have to interact with a large number of people to get it all set up. The merge may cause problems with other PLIP's coming in. During the merge phase you must be prepared to help out with all the features and bugs that arise.

If all went as planned the next Plone release will carry on with your PLIP in it. You'll be expected to help out with that feature after it's been released (within reason).

Troubleshooting

Buildout Issues Buildout can be frustrating for those unfamiliar with parsing through autistic robot language. Fear not! These errors are almost always a quick fix and a little bit of understanding goes a long ways.

Errors Running bootstrap.py You may not even get to running buildout and then you will already have an error. Let's take this one for example:

```
...
 File "/usr/local/lib/python2.6/site-packages/distribute-0.6.13-py2.6.egg/pkg_resources.py", line 556
    raise VersionConflict(dist,req) # XXX put more info here
 pkg_resources.VersionConflict: (zc.buildout 1.5.1 (/usr/local/lib/python2.6/site-packages/zc.buildou
```

You may think the buildout god is angry because it's been MONTHS since you've made a human sacrifice to her but be strong and follow along. Buildout has simply noticed that the version of buildout required by the bootstrap.py file you are trying to run does not match the version of buildout in your python library. In the error above, your system has buildout 1.5.1 installed and the bootstrap.py file wants to run with 1.5.2.

To fix, you have a couple options. First, you can force buildout to run with the version you already have installed by invoking the version tag. This tells your [Plone] bootstrap.py file to play nicely with the version that you already have installed. In the case of the error pasted above, that would be:

```
> python bootstrap.py --version=1.5.1
```

I personally know that versions 1.4.4, 1.5.1, and 1.5.2 all work this way.

The other option is to delete your current egg and force the upgrade. In the case of the error above, all you need to do is delete the egg the system currently has. eg:

```
> rm -rf /usr/local/lib/python2.6/site-packages/zc.buildout-1.5.1-py2.6.egg
```

When you rerun bootstrap, it will look for the buildout of the egg, note that there isn't one, and then go fetch a new egg in the version that it wants for you.

Do one of those, say two hail marys, and re-run bootstrap. Tada!

One other thing of note is that running bootstrap effectively ties that python executable and all of its libraries to your buildout. If you have several python installs and want to switch which python is tied to your buildout, simply rerun bootstrap.py with the new python (and then rerun buildout). You may get the same error above again but now that you know how to fix it, you can spend that time drinking beer instead of smashing your keyboard.

Hooray!

When Mr. Developer is Unhappy mr.developer is never unhappy, except when it is. Although this technically isn't a buildout issues, it happens when running buildout so I'm putting it under buildout issues.

When working with the dev instance, especially with all the moving back and forth between github and svn, you may have an old copy of a src package. The error looks like:

```
mr.developer: Can't update package 'Products.CMFPlone' because its URL doesn't match.
```

As long as you don't have any pending commits, you just need to remove the package from src/ and it will recheck it out for you when it updates.

You can also get such fun errors as:

```
Link to http://sphinx.pocoo.org/ ***BLOCKED*** by --allow-hosts
```

These are ok to ignore IF and ONLY IF the lines following it say:

```
Getting distribution for 'Sphinx==1.0.7'.
Got Sphinx 1.0.7.
```

If buildout ends with warning you that some packages could not be downloaded, then chances are that package wasn't downloaded. This is bad and could cause all sorts of whack out errors when you start or try to run things because it never actually downloaded the package.

There are two ways to get this error to go away. The first is to delete all instances of host filtering. Comb through all the files and delete any lines which say "allow-hosts =" and "allow-hosts +=". In theory, by restricting which hosts you download from, buildout will go faster. Whether that actually happens or not I can not judge. The point is that they are safely deletable.

The second option is to allow the host that it is pointing to by adding something like this to your .cfg:

```
allow-hosts += sphinx.pocoo.org
```

Again, this is only necessary if the package wasn't found in the end.

Hooray!

mr.developer Path Errors `ERROR: You are not in a path which has mr.developer installed (.mr.developer.cfg not found).`

When running any `./bin/develop` command.

To fix, simply do:

```
``ln -s plips/.mr.developer.cfg``
```

Other Random Issues

Dirty Packages "ERROR: Can't update package '[Some package]', because it's dirty."

Fix mr.developer is complaining because a file has been changed/added, but not committed.

Use `bin/develop update --force.` Adding `*.pyc *~.nib *.egg-info .installed.cfg *.pt.py *.cpt.py *.zpt.py *.html.py *.egg` to your subversion config's global-ignores has been suggested as a more permanent solution.

No module named zope 2 `ImportError: No module named Zope2" when building using a PLIP cfg file.`

Appears to not actually be the case. Delete 'mkzopeinstance.py' from bin/ and rerun buildout to correct this if you're finding it irksome.

Can't open file '/Startup/run.py' Two possible fixes, you are using Python 2.4 by mistake, so use 2.6 instead. Or, you may need to make sure you run 'bin/buildout . . . ' after 'bin/develop . . . '. Try removing parts/, *bin/*, .installed.cfg, then re-bootstrap and re-run buildout, develop, buildout.

Missing PIL pil.cfg is include within this buildout to aid in PIL installation. Run bin/buildout -c pil.cfg to install. This method does not work on Windows, so we're unable to run it by default.

Modified Egg Issues bin/develop status is showing that the Products.CMFActionIcons egg has been modified, but I haven't touched it. And this is preventing bin/develop up from updating all the eggs.

Fix Edit ~/.subversion/config and add eggtest*.egg to the list of global-ignores

Como atualizar estes documentos Estes documentos são armazenados com o buildout coredev no github em /docs. Para atualizá-los, por favor, faça o checkout do buildout coredev e atualize. Para verificar as alterações localmente, execute novamente o buildout e depois:

```
> cd docs/build
> make html
```

O Sphinx irá gerar um diretório com arquivos HTML, que você pode conferir com o seu navegador. Por favor, verifique todos os avisos e erros antes de comitar para garantir que os documentos continuam válidos.

Others

These are some documents using as reference for this documentation.

Mr. Developer Esse buildout usa o mr.developer para gerenciar o desenvolvimento do pacote. Veja http://pypi.python.org/pypi/mr.developer para maiores informações ou execute 'bin/develop help' para obter uma lista de comandos disponíveis.

A forma mais comum de obter as últimas atualizações é:

$ git pull $ bin/develop rb

Isto lhe trará a versão mais recente da configuração do coredev, faça o checkout, atualize todos os pacotes do Subversion na pasta src e execute buildout para configurar tudo.

De vez em quando você pode verificar se existem modificações não comitadas nos repositórios baixados pelo mr.developer:

$ bin/develop st

Se alguma linha for impressa com um ponto de interrogação na frente, você pode limpar a lista com o seguinte comando:

$ bin/develop purge

Isso irá remover os pacotes desnecessários de src/, já que foram substituídos por versões de eggs mais adequadas.

Revisando PLIP's

Expectativas Uma boa revisão de PLIP leva aproximadamente 4 horas, então se programe corretamente. Quando terminar, se tiver acesso ao núcleo, comite a revisão no diretório de PLIP's e referencie a PLIP na mensagem do seu commit. Se não tiver acesso, anexe sua revisão ao ticket aberto da PLIP.

Configurando o ambiente Siga as instruções em [wiki:DevelopmentEnvironment setting up a development environment], "Obtendo o Código". Você deverá fazer o checkout do branch ao qual a PLIP está vinculada. Ao invés de rodar o buildout usando o arquivo de buildout padrão, você deverá rodar a configuração específica para a PLIP:

```
> ./bin/buildout -c plips/plipXXXX.cfg
```

Revisão de funcionalidade Existem várias coisas que podem ser abordadas em uma revisão de PLIP, dependendo da sua natureza. A lista abaixo não é, absolutamente, uma lista completa, mas um ponto de partida sobre o que pode ser observado em uma revisão:

Geral

- A PLIP realmente faz o que os implementadores propuseram? Existem variações incompletas?
- Ocorreram erros durante a execução do buildout? As migrações funcionaram?
- Os erros e mensagens de status fazem sentido? Estão internacionalizados corretamente?
- Existem considerações sobre a performance? O implementador as observou?

Erros

- Existem erros? Nada é muito grande ou muito pequeno.
- Os campos manipulam dados estranhos? E situações como strings em campos de data ou campos obrigatórios deixados em branco?
- A validação não está pouco exigente nem exigente demais?

Qestões de usabilidade

- A implementação está usável?
- Como usuários finais iniciantes reagirão à mudança?
- A PLIP necessita de uma revisão de usabilidade? Se você considerar que sim, altere o estado para "please review" e adicione uma nota nos comentários.
- A PLIP está consistente com o resto do Plone? Por exemplo, se existe uma configuração no painel de controle, o novo formulário se adequa aos outros painéis?
- Tudo está fluindo bem para usuários iniciantes e avançados? Existem fluxos que parecem estranhos?
- Existem permissões novas e elas funcionam apropriadamente? A atribuição de papéis (roles) faz sentido?

Questões de documentação

- A documentação correspondente é suficiente para usuários finais, seja ele desenvolvedor ou usuário Plone?
- A mudança está bem documentada?

Reporte erros/solicitações no Trac, como faria com qualquer outro erro do Plone. Referencie a PLIP no erro, atribua a seu desenvolvedor e adicione uma tag para a PLIP como plip-xxx, pois fica mais fácil do desenvolvedor encontrar, se precisar. Também coloque prioridade no ticket. A PLIP não será mesclada até que os erros sejam corrigidos.

Revisão de Código

Python

- O código é possível de manter?
- O código está documentado corretamente?
- O código segue os padrões da PEP8? Quanto?
- Está importando módulos obsoletos?

Javascript

- O javascript segue os padrões utilizados? Veja a referência em http://plone.org/documentation/manual/developer-manual/client-side-functionality-javascript/javascript-standards/referencemanual-all-pages
- O javascript funciona em todas os navegadores suportados atualmente? Como está o desempenho?

ME/TAL

- A PLIP usa views apropriadas e evita implementar muita lógica?
- Existe algum loop no código que pode ter, potencialmente, problema de desempenho?
- Existem linhas de código obsoletas ou no estilo antigo, como DateTime?
- O HTML renderizado é compatível com as normas? Os ids e classes são usadas apropriadamente?

Exemplos de revisões de PLIP

- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-davisagli.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip10886-review-cah190.txt
- https://svn.plone.org/svn/plone/buildouts/plone-coredev/branches/4.1/plips/plip9352-review-rossp.txt